

Первый курс, осенний семестр 2017/18

Конспект лекций по алгоритмам

Собрано 23 июля 2018 г. в 17:51

Содержание

1. Разбор теста	1
1.1. Разбор основных задач	1
1.2. Разбор дополнительных задач	2
2. Асимптотика	3
2.1. O -обозначения	3
2.2. Рекуррентности и Карацуба	4
2.3. Теоремы о рекуррентных соотношениях	6
2.4. Доказательства по индукции	7
2.5. Числа Фибоначчи	7
2.6. O -обозначения через пределы	7
2.7. Замена сумм на интегралы	8
2.8. Примеры по теме асимптотики	9
2.9. Сравнение асимптотик	10
3. Структуры данных	11
3.1. C++	12
3.2. Неасимптотические оптимизации	13
3.3. Частичные суммы	14
3.4. Массив	14
3.5. Двусвязный список	14
3.6. Односвязный список	15
3.7. Список на массиве	15
3.8. Вектор (расширяющийся массив)	16
3.9. Стек, очередь, дек	16
3.10. Очередь, стек и дек с минимумом	17
4. Структуры данных	17
4.1. Амортизационный анализ	18
4.2. Разбор арифметических выражений	19
4.3. Бинарный поиск	20
4.3.1. Обыкновенный	20
4.3.2. По предикату	20
4.3.3. Вещественный, корни многочлена	21
4.4. Два указателя и операции над множествами	21
4.5. Хеш-таблица	22
4.5.1. На списках	22
4.5.2. С открытой адресацией	22
4.5.3. C++	23

5. Структуры данных	24
5.1. Избавляемся от амортизации	24
5.1.1. Вектор (решаем проблему, когда случится)	24
5.1.2. Вектор (решаем проблему заранее)	24
5.1.3. Хеш-таблица	24
5.1.4. Очередь с минимумом через два стека	25
5.2. Бинарная куча	26
5.2.1. GetMin, Add, ExtractMin	26
5.2.2. Обратные ссылки и DecreaseKey	27
5.2.3. Build, HeapSort	27
5.3. Аллокация памяти	28
5.3.1. Стек	28
5.3.2. Список	28
5.3.3. Куча и хеш-таблица	29
5.3.4. Куча	29
5.4. Пополняемые структуры	30
5.4.1. Ничего → Удаление	30
5.4.2. Поиск → Удаление	30
5.4.3. Add → Merge	30
5.4.4. Build → Add	31
5.4.5. Build → Add, Del	31
5.5. Два указателя и алгоритм Mo	32
6. Сортировки	33
6.1. Квадратичные сортировки	33
6.2. Оценка снизу на время сортировки	34
6.3. Быстрые сортировки	34
6.3.1. CountSort (подсчётом)	34
6.4. Решение задачи по пройденным темам	34
6.4.1. MergeSort (слиянием)	35
6.4.2. QuickSort (быстрая)	35
6.4.3. Сравнение сортировок	36
7. Сортировки (продолжение)	37
7.1. Quick Sort	37
7.1.1. Оценка времени работы	37
7.1.2. Introsort'97	38
7.2. Порядковые статистики	39
7.2.1. Одноветочный QuickSort	39
7.2.2. Детерминированный алгоритм	39
7.2.3. C++	40
7.3. Integer sorting	40
7.3.1. Count sort	40
7.3.2. Radix sort	40
7.3.3. Bucket sort	40
7.4. Kirkpatrick'84 sort	42

8. Кучи	42
8.1. Min-Max Heap (Atkison'86)	43
8.2. Leftist Heap (Clark'72)	44
8.3. Skew Heap (Tarjan'86)	44
8.4. Спско-куча	45
8.5. Van Embde Boas'75 trees	45
9. Кучи	46
9.1. Нижняя оценка на построение бинарной кучи	47
9.2. Биномиальная куча (Vuillemin'78)	48
9.2.1. Основные понятия	48
9.2.2. Операции с биномиальной кучей	49
9.2.3. Add и Merge за $\mathcal{O}(1)$	49
9.3. Куча Фибоначчи (Fredman, Tarjan'84)	49
9.3.1. Фибоначчиевы деревья	51
9.3.2. Завершение доказательства	51
10. Динамическое программирование	52
10.1. Базовые понятия	52
10.1.1. Условие задачи	52
10.1.2. Динамика назад	52
10.1.3. Динамика вперёд	53
10.1.4. Ленивая динамика	53
10.2. Ещё один пример	54
10.3. Восстановление ответа	54
10.4. Графовая интерпретация	55
10.5. Checklist	56
10.6. Рюкзак	56
10.6.1. Формулировка задачи	56
10.6.2. Решение динамикой	56
10.6.3. Оптимизируем память	56
10.6.4. Добавляем <code>bitset</code>	57
10.6.5. Восстановление ответа с линейной памятью	57
10.7. Квадратичные динамики	58
10.8. Оптимизация памяти для НОП	59
10.8.1. Храним биты	59
10.8.2. Алгоритм Хиршберга (по wiki)	59
10.8.3. Оценка времени работы Хиршберга	59
10.8.4. Алгоритм Хиршберга (улучшенный)	60
10.8.5. Область применения идеи Хиршберга	60
11. Динамическое программирование (часть 2)	61
11.1. <code>bitset</code>	61
11.1.1. Рюкзак	61
11.2. НВП за $\mathcal{O}(n \log n)$	61
11.3. Задача про погрузку кораблей	62
11.3.1. Измельчение перехода	62

11.3.2. Использование пары, как функции динамики	63
11.4. Рекуррентные соотношения	63
11.4.1. Пути в графе	64
11.5. Задача о почтовых отделениях	64
11.6. Оптимизация Кнута	65
11.7. Оптимизация методом “разделяй и властвуй”	66
11.8. Стресс тестирование	66
12. Динамическое программирование (часть 3)	66
12.1. Динамика по подотрезкам	67
12.2. Комбинаторика	67
12.3. Работа с множествами	68
12.4. Динамика по подмножествам	69
12.5. Гамильтоновы путь и цикл	70
12.6. Вершинная покраска	70
12.7. Вершинная покраска: решение за $\mathcal{O}(3^n)$	71
13. Динамическое программирование (часть 4)	71
13.1. Вершинная покраска: решение за $\mathcal{O}(2.44^n)$	72
13.2. Set cover	73
13.3. Bit reverse	73
13.4. Meet in the middle	73
13.4.1. Количество клик в графе за $\mathcal{O}(2^{n/2})$	73
13.4.2. Рюкзак за $\mathcal{O}(2^{n/2}n)$	74
13.5. Динамика по скошенному профилю	75
14. Графы и поиск в глубину	76
14.1. Определения	77
14.2. Хранение графа	78
14.3. Поиск в глубину	79
14.4. Топологическая сортировка	79
14.5. Компоненты сильной связности	79
15. Поиск в глубину (часть 2)	80
15.1. Сильная связность	80
15.2. Рёберная двусвязность	80
15.3. Вершинная двусвязность	81
15.4. Эйлеровость	82
15.5. 2-SAT	83
15.5.1. Решение 2-SAT за $\mathcal{O}(n + m)$	84
15.5.2. Решение 3-SAT и 3-List-Coloring	85

Лекция #1: Разбор теста

4 сентября

1.1. Разбор основных задач

• Задача #1

- (a) $O(n^2)$ — сортировка вставками, выбором, пузырьком
- (a) $O(n \log n)$ — merge sort, quick sort, heap sort
- (b) $O(n + m)$ — сортировка подсчетом.

```
1 vector<int> cnt(M, 0); // M нулей
2 for (i = 0; i < n; i++) // O(n)
3     cnt[a[i]] += 1;
4 for (i = 0; i < m; i++) // O(m)
5     for (j = 0; j < cnt[i]; j++) // O(n) в сумме по всем i
6         print(i); // выводим cnt[i] раз число i
```

- (b) $O(n), O(m)$ — формально **не верно**, правильно $O(n + m)$.
- (b) $O(n + n \log_n m)$ — цифровая сортировка (radix sort, digital sort).
- (b) Дополнительные баллы всем, кто вспомнил про длинные числа.
- (c) MergeSort работает за $\Theta(n \log n)$ всегда.

• Задача #2

- (a) Дек, реализация на массиве, $O(1)$.
- (b) Массив: $O(1)$. Код удаления: `i = rand() % n; swap(a[i], a[n-1]); return a[--n];`
- (c) Двусвязный список + массив, $O(1)$.
Node p[]; p[i] — позиция в списке элемента, добавленного в i -й момент времени
- (d) Хеш таблица, рандомизированное $O(1)$

• Задача #3a

```
1 max = a[0];
2 for (i = 1; i < n; i++) {
3     if (a[i] + max > S) return true;
4     if (a[i] > max) max = a[i];
5 }
6 return false;
```

• Задача #36 . Решение, за $O(\sqrt{n})$:

```
1 for (a = 1; a * a <= n; a++) {
2     b = sqrt(n - a * a);
3     res += (a * a + b * b == n);
4 }
```

Решение, за $O(\sqrt{n})$ элементарных арифметических операций с целыми числами:

```
1 int m = sqrt(n / 2), b = sqrt(n);
2 for (int a = 1; a <= m; a++) {
3     while ((tmp = a * a + b * b) > n) --b; // 2(n/2)^{1/2} умножений
4     if (tmp == n) res += (a == b ? 1 : 2);
5 }
```

• Задача #4а

Одна из возможных идей — предподсчет за $\mathcal{O}(n^2)$, можно положить все элементы матрицы в хеш-таблицу и за $\mathcal{O}(1)$ отвечать на поступающие запросы. Если предподсчет запрещен, то на один запрос можно отвечать за время $\mathcal{O}(n)$:

```

1 i = n - 1, j = 0;
2 while (i >= 0 && j < n) {
3     if (a[i][j] == x) return 1;
4     (a[i][j] < x) ? ++j : --i;
5 }
```

Заметим, что без предподсчета быстрее чем за $\mathcal{O}(n)$ на запрос не ответить.

• Задача #4б

Существуют решения за $\mathcal{O}(n)$, это или алгоритм Манакера (аналог Z -функции), или построение дерева палиндромов. Оба алгоритма просты в реализации и встретятся в курсе позднее. Здесь описаны лишь более простые решения. Итак, решения за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$:

1. У каждого палиндрома есть центр. Центр находится или в позиции i , или между позициями i и $(i + 1)$. Центр можно перебрать.
2. Решим задачу для фиксированного центра x . Если есть палиндром с центром в x и длиной L , то есть и палиндромы с центром в x и длинами $L - 2, L - 4, \dots$. Поэтому достаточно найти максимальное L .
3. Поиск максимального L за линейное время: `while (s[x+L] == s[x-L]) L++;`
4. Поиск максимального L за $\mathcal{O}(\log n)$: бинарный поиск, а внутри сравнение подстрок за $\mathcal{O}(1)$ с помощью полиномиальных хешей.

1.2. Разбор дополнительных задач

• Задача #8а

Рассмотрим все числа, как двоичные строки и положим в бор (trie). Ближе к корню старший бит. В каждой вершине бора будем хранить размер поддерева. Теперь при XOR достаточно для каждого уровня бора поменять порядок детей, это работает за глубину.

• Задача #8б

Пусть нужно перевести число X из $n = 2^k$ цифр в двоичную систему – найти $F(X)$.

Обозначим число из $n/2$ старших цифр X_0 , младшие цифры – X_1 :

$$X = X_0 10^{n/2} + X_1 \Rightarrow F(X) = F(X_0)F(10^{n/2}) + F(X_1).$$

Для умножения двоичных чисел $F(X_0)$ и $F(10^{n/2})$ воспользуемся функцией за $\mathcal{O}(n \log n)$.

Все $F(10^{2^k})$ можно предподсчитать за $\mathcal{O}(n \log n)$ в сумме: $F(10^{2^k}) = F(10^{2^{k-1}})^2$.

Итого время работы $T(n) = 2T(n/2) + n \log n = \Theta(n \log^2 n)$.

Лекция #2: Асимптотика

4 сентября

2.1. \mathcal{O} -обозначения

Рассмотрим функции $f, g: \mathbb{N} \rightarrow \mathbb{R}^{>0}$.

Def 2.1.1. $f = \Theta(g)$ $\Leftrightarrow \exists N > 0, C_1 > 0, C_2 > 0: \forall n \geq N, C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$

Def 2.1.2. $f = \mathcal{O}(g)$ $\Leftrightarrow \exists N > 0, C > 0: \forall n \geq N, f(n) \leq C \cdot g(n)$

Def 2.1.3. $f = \Omega(g)$ $\Leftrightarrow \exists N > 0, C > 0: \forall n \geq N, f(n) \geq C \cdot g(n)$

Def 2.1.4. $f = o(g)$ $\Leftrightarrow \forall C > 0 \exists N > 0: \forall n \geq N, f(n) \leq C \cdot g(n)$

Def 2.1.5. $f = \omega(g)$ $\Leftrightarrow \forall C > 0 \exists N > 0: \forall n \geq N, f(n) \geq C \cdot g(n)$

Понимание Θ : “равны с точностью до константы”, “асимптотически равны”.

Понимание \mathcal{O} : “не больше с точностью до константы”, “асимптотически не больше”

Понимание o : “асимптотически меньше”, “для сколь угодно малой константы не больше”

Θ	\mathcal{O}	Ω	o	ω
$=$	\leq	\geq	$<$	$>$

Замечание 2.1.6. $f = \Theta(g) \Leftrightarrow g = \Theta(f)$

Замечание 2.1.7. $f = \mathcal{O}(g), g = \mathcal{O}(f) \Leftrightarrow f = \Theta(g)$

Замечание 2.1.8. $f = \Omega(g) \Leftrightarrow g = \mathcal{O}(f)$

Замечание 2.1.9. $f = \omega(g) \Leftrightarrow g = o(f)$

Замечание 2.1.10. $f = \mathcal{O}(g), g = \mathcal{O}(h) \Rightarrow f = \mathcal{O}(h)$

Замечание 2.1.11. Обобщение: $\forall \beta \in \{\mathcal{O}, o, \Theta, \Omega, \omega\}: f = \beta(g), g = \beta(h) \Rightarrow \boxed{f = \beta(h)}$

Замечание 2.1.12. $\forall C > 0 \quad Cf = \Theta(f)$

Докажем для примера 2.1.6.

Доказательство. $C_1 g(n) \leq f(n) \leq C_2 g(n) \Rightarrow \frac{1}{C_2} f(n) \leq g(n) \leq \frac{1}{C_1} g(n) \leq f(n)$ ■

Упражнение 2.1.13. $f = \mathcal{O}(\Theta(\mathcal{O}(g))) \Rightarrow f = \mathcal{O}(g)$

Упражнение 2.1.14. $f = \Theta(o(\Theta(\mathcal{O}(g)))) \Rightarrow f = o(g)$

Упражнение 2.1.15. $f = \Omega(\omega(\Theta(g))) \Rightarrow f = \omega(g)$

Упражнение 2.1.16. $f = \Omega(\Theta(\mathcal{O}(g))) \Rightarrow f$ может быть любой функцией

Lm 2.1.17. $g = o(f) \Rightarrow f \pm g = \Theta(f)$

Доказательство. $g = o(f) \exists N: \forall n \geq N \quad g(n) \leq \frac{1}{2} f(n) \Rightarrow \frac{1}{2} f(n) \leq f(n) \pm g(n) \leq \frac{3}{2} f(n)$ ■

Lm 2.1.18. $n^k = o(n^{k+1})$

Доказательство. $\forall C \forall n \geq C \quad n^{k+1} \geq C \cdot n^k$ ■

Lm 2.1.19. $P(x) = \Theta(x^{\deg P})$ при положительном старшем коэффициенте.

Доказательство. $P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_k x^k$. По леммам 2.1.12, 2.1.18 имеем, что все слагаемые кроме $a_k x^k$ являются $o(x^{\deg P})$. Поэтому по лемме 2.1.17 вся сумма является $\Theta(x^k)$. ■

2.2. Рекуррентности и Карацуба

• Алгоритм умножения чисел в столбик

Рассмотрим два многочлена $A(x) = 5 + 4x + 3x^2 + 2x^3 + x^4$ и $B(x) = 9 + 8x + 7x^2 + 6x^3$.

Запишем массивы $a[] = \{5, 4, 3, 2, 1\}$, $b[] = \{9, 8, 7, 6\}$.

```
1 for (i = 0; i < an; i++) // an = 5
2   for (j = 0; j < bn; j++) // bn = 4
3     c[i + j] += a[i] * b[j];
```

Мы получили в точности коэффициенты многочлена $C(x) = A(x)B(x)$.

Теперь рассмотрим два числа $A = 12345$ и $B = 6789$, запишем те же массивы и сделаем:

```
1 // Перемножаем числа без переносов, как многочлены
2 for (i = 0; i < an; i++) // an = 5
3   for (j = 0; j < bn; j++) // bn = 4
4     c[i + j] += a[i] * b[j];
5 // Делаем переносы, массив c = [45, 76, 94, 100, 70, 40, 19, 6, 0]
6 for (i = 0; i < an + bn; i++)
7   if (c[i] >= 10)
8     c[i + 1] += c[i] / 10, c[i] %= 10;
9 // Массив c = [5, 0, 2, 0, 1, 8, 3, 8, 0], ответ = 83810205
```

Данное умножение работает за $\Theta(nm)$, или $\Theta(n^2)$ в случае $n = m$.

Следствие 2.2.1. Чтобы умножать длинные числа достаточно уметь умножать многочлены.

Многочлены мы храним, как массив коэффициентов. При программировании умножения, нам важно знать не степень многочлена d , а длину этого массива $n = d + 1$.

• Алгоритм Карацубы

Чтобы перемножить два многочлена (или два длинных целых числа) $A(x)$ и $B(x)$ из n коэффициентов каждый, разделим их на части по $k = \frac{n}{2}$ коэффициентов — A_1, A_2, B_1, B_2 .

Заметим, что $A \cdot B = (A_1 + x^k A_2)(B_1 + x^k B_2) = A_1 B_1 + x^k (A_1 B_2 + A_2 B_1) + x^{2k} A_2 B_2$.

Если написать рекурсивную функцию умножения, то получим время работы:

$$T_1(n) = 4T_1\left(\frac{n}{2}\right) + \Theta(n)$$

Из последующей теоремы мы сделаем вывод, что $T_1(n) = \Theta(n^2)$. Алгоритм можно улучшить, заметив, что $A_1 B_2 + A_2 B_1 = (A_1 + A_2)(B_1 + B_2) - A_1 B_1 - A_2 B_2$, где вычитаемые величины уже посчитаны. Итого три умножения вместо четырёх:

$$T_2(n) = 3T_2\left(\frac{n}{2}\right) + \Theta(n)$$

Из последующей теоремы мы сделаем вывод, что $T_2(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585\dots})$.

Данный алгоритм применим и для умножения многочленов, и для умножения чисел.

Псевдокод алгоритма Карацубы для умножения многочленов:

```
1 //  $n = 2^k$ ,  $c(w) = a(w)*b(w)$ 
2 Mul(n, a, b) {
3   if  $n == 1$ : return {a[0] * b[0]}
4   a --> a1, a2
5   b --> b1, b2
6   x = Mul(n / 2, a1, b1)
7   y = Mul(n / 2, a2, b2)
8   z = Mul(n / 2, a1 + a2, b1 + b2)
9   // Умножение на  $w^i$  - сдвиг массива на  $i$  вправо
10  return x + y *  $w^n$  + (z - x - y) *  $w^{\{n/2\}}$ ;
11 }
```

Чтобы умножить числа, сперва умножим их как многочлены, затем сделаем переносы.

2.3. Теоремы о рекуррентных соотношениях

Теорема 2.3.1. *Мастер Теорема* (теорема о простом рекуррентном соотношении)

Пусть $T(n) = aT(\frac{n}{b}) + f(n)$, где $f(n) = n^c$. При этом $a > 0, b > 1, c \geq 0$. Определим глубину рекурсии $k = \log_b n$. Тогда верно одно из трёх:

$$\begin{cases} T(n) = \Theta(a^k) = \Theta(n^{\log_b a}) & a > b^c \\ T(n) = \Theta(f(n)) = \Theta(n^c) & a < b^c \\ T(n) = \Theta(k \cdot f(n)) = \Theta(n^c \log n) & a = b^c \end{cases}$$

Доказательство. Раскроем рекуррентность:

$$T(n) = f(n) + aT(\frac{n}{b}) = f(n) + af(\frac{n}{b}) + a^2f(\frac{n}{b^2}) + \dots = n^c + a(\frac{n}{b})^c + a^2(\frac{n}{b^2})^c + \dots$$

Тогда $T(n) = f(n)(1 + \frac{a}{b^c} + (\frac{a}{b^c})^2 + \dots + (\frac{a}{b^c})^k)$. При этом в сумме $k + 1$ слагаемых.

Обозначим $q = \frac{a}{b^c}$ и оценим сумму $S(q) = 1 + q + \dots + q^k$.

Если $q = 1$, то $S(q) = k + 1 = \log_b n + 1 = \Theta(\log_b n) \Rightarrow T(n) = \Theta(f(n) \log n)$.

Если $q < 1$, то $S(q) = \frac{1 - q^{k+1}}{1 - q} = \Theta(1) \Rightarrow T(n) = \Theta(f(n))$.

Если $q > 1$, то $S(q) = q^k + \frac{q^k - 1}{q - 1} = \Theta(q^k) \Rightarrow T(n) = \Theta(a^k (\frac{n}{b^k})^c) = \Theta(a^k)$. ■

Теорема 2.3.2. *Обобщение Мастер Теоремы*

Мастер Теорема верна и для $f(n) = n^c \log^d n$

$T(n) = aT(\frac{n}{b}) + n^c \log^d n$. При $a > 0, b > 1, c \geq 0, d \geq 0$.

$$\begin{cases} T(n) = \Theta(a^k) = \Theta(n^{\log_b a}) & a > b^c \\ T(n) = \Theta(f(n)) = \Theta(n^c \log^d n) & a < b^c \\ T(n) = \Theta(k \cdot f(n)) = \Theta(n^c \log^{d+1} n) & a = b^c \end{cases}$$

Без доказательства. ■

Теорема 2.3.3. *Об экспоненциальном рекуррентном соотношении*

Пусть $T(n) = \sum b_i T(n - a_i)$. При этом $a_i > 0, b_i > 0, \sum b_i > 1$.

Тогда $T(n) = \Theta(\alpha^n)$, при этом $\alpha > 1$ и является корнем уравнения $1 = \sum b_i \alpha^{-a_i}$, его можно найти бинарным поиском.

Доказательство. Предположим, что $T(n) = \alpha^n$, тогда $\alpha^n = \sum b_i \alpha^{n-a_i} \Leftrightarrow 1 = \sum b_i \alpha^{-a_i} = f(\alpha)$.

Теперь нам нужно решить уравнение $f(\alpha) = 1$ для $\alpha \in [1, +\infty)$.

Если $\alpha = 1$, то $f(\alpha) = \sum b_i > 1$, если $\alpha = +\infty$, то $f(\alpha) = 0 < 1$. Кроме того $f(\alpha) \searrow [1, +\infty)$.

Получаем, что на $[1, +\infty)$ есть единственный корень уравнения $1 = f(\alpha)$ и его можно найти бинарным поиском.

Мы показали, откуда возникает уравнение $1 = \sum b_i \alpha^{-a_i}$. Доказали, что у него $\exists!$ корень α .

Теперь докажем по индукции, что $T(n) = \mathcal{O}(\alpha^n)$ (оценку сверху) и $T(n) = \Omega(\alpha^n)$ (оценку снизу). Доказательства идентичны, покажем $T(n) = \mathcal{O}(\alpha^n)$. База индукции:

$$\exists C: \forall n \in B = [1 - \max_i a_i, 1] \quad T(n) \leq C \alpha^n$$

Переход индукции:

$$T(n) = \sum b_i T(n - a_i) \stackrel{\text{по индукции}}{\leq} C \sum b_i \alpha^{n-a_i} \stackrel{(*)}{=} C \alpha^n$$

(*) Верно, так как α – корень уравнения. ■

2.4. Доказательства по индукции

Lm 2.4.1. Доказательство по индукции

Есть простой метод решения рекуррентных соотношений: угадать ответ, доказать его по индукции. Рассмотрим на примере $T(n) = \max_{x=1..n-1} (T(x) + T(n-x) + x(n-x))$.

Докажем, что $T(n) = \mathcal{O}(n^2)$, для этого достаточно доказать $T(n) \leq n^2$:

База: $T(1) = 1 \leq 1^2$.

Переход: $T(n) \leq \max_{x=1..n-1} (x^2 + (n-x)^2 + x(n-x)) \leq \max_{x=1..n-1} (x^2 + (n-x)^2 + 2x(n-x)) = n^2$

• Примеры по теме рекуррентные соотношения

1. $T(n) = T(n-1) + T(n-1) + T(n-2)$.

Угадаем ответ 2^n , проверим по индукции: $2^n = 2^{n-1} + 2^{n-1} + 2^{n-2}$.

2. $T(n) = T(n-3) + T(n-3) \Rightarrow T(n) = 2T(n-3) = 4T(n-6) = \dots = 2^{n/3}$

3. $T(n) = T(n-1) + T(n-3)$. Применяем 2.3.3, получаем $1 = \alpha^{-1} + \alpha^{-3}$, находим α бинарным поиском, получаем $\alpha = 1.4655\dots$

2.5. Числа Фибоначчи

Def 2.5.1. $f_1 = f_0 = 1, f_i = f_{i-1} + f_{i-2}$. f_n — n -е число Фибоначчи.

• Оценки снизу и сверху

$f_n = f_{n-1} + f_{n-2}$, рассмотрим $g_n = g_{n-1} + g_{n-1}$, $2^n = g_n \geq f_n$.

$f_n = f_{n-1} + f_{n-2}$, рассмотрим $g_n = g_{n-2} + g_{n-2}$, $2^{n/2} = g_n \leq f_n$.

Воспользуемся 2.3.3, получим $1 = \alpha^{-1} + \alpha^{-2} \Leftrightarrow \alpha^2 - \alpha - 1 = 0$, получаем $\alpha = \frac{\sqrt{5}+1}{2} \approx 1.618$.

$f_n = \Theta(\alpha^n)$.

2.6. \mathcal{O} -обозначения через пределы

Def 2.6.1. $f = o(g)$ Определение через предел: $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$

Def 2.6.2. $f = \mathcal{O}(g)$ Определение через предел: $\overline{\lim}_{n \rightarrow +\infty} \frac{f(n)}{g(n)} < \infty$

Здесь необходимо пояснение: $\overline{\lim}_{n \rightarrow +\infty} f(n) = \lim_{n \rightarrow +\infty} (\sup_{x \in [n, +\infty]} f(x))$, где \sup — верхняя грань.

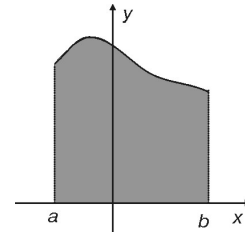
Lm 2.6.3. Определения \mathcal{O} эквивалентны

Доказательство. Вспомним, что речь о положительных функциях f и g .

Распишем предел по определению: $\forall C > 0 \quad \exists N \quad \forall n \geq N \quad \frac{f(n)}{g(n)} \leq C \Leftrightarrow f(n) \leq Cg(n)$. ■

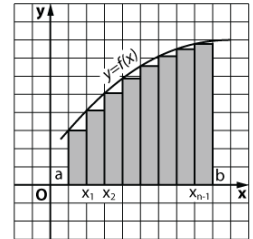
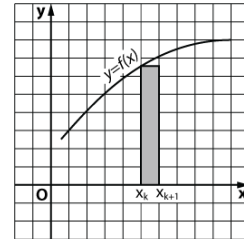
2.7. Замена сумм на интегралы

Def 2.7.1. Определённый интеграл $\int_a^b f(x)dx$ положительной функции $f(x)$ – площадь под графиком f на отрезке $[a..b]$.



Lm 2.7.2. $\forall f(x) \nearrow [a..a+1] \Rightarrow f(a) \leq \int_a^{a+1} f(x)dx \leq f(a+1)$

Lm 2.7.3. $\forall f(x) \nearrow [a..b+1] \Rightarrow \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x)dx$



Доказательство. Сложили неравенства из 2.7.2 ■

Lm 2.7.4. $\forall f(x) \nearrow [a..b], f > 0 \Rightarrow \int_a^b f(x)dx \leq \sum_{i=a}^b f(i)$

Доказательство. Сложили неравенства из 2.7.2, выкинули $[a-1, a]$ из интеграла. ■

Теорема 2.7.5. Замена суммы на интеграл #1

$$\forall f(x) \nearrow [1..\infty), f > 0, S(n) = \sum_{i=1}^n f(i), I_1(n) = \int_1^n, I_2(n) = \int_1^{n+1}, I_1(n) = \Theta(I_2(n)) \Rightarrow S(n) = \Theta(I_1(n))$$

Доказательство. Из лемм 2.7.3 и 2.7.4 имеем $I_1(n) \leq S(n) \leq I_2(n)$.

$$C_1 I_1(n) \leq I_2(n) \leq C_2 I_1(n) \Rightarrow I_1(n) \leq S(n) \leq I_2(n) \leq C_2 I_1(n)$$

Теорема 2.7.6. Замена суммы на интеграл #2

$$\forall f(x) \nearrow [a..b], f > 0 \quad \int_a^b f(x)dx \leq \sum_{i=a}^b f(i) \leq f(b) + \int_a^b f(x)dx$$

Доказательство. Первое неравенство – лемма 2.7.3. Второе – 2.7.4, применённая к $\sum_{i=a}^{b-1}$. ■

Следствие 2.7.7. Для убывающих функций два последних факта тоже верны. Во втором ошибкой будет не $f(b)$, а $f(a)$, которое теперь больше.

• Как считать интегралы?

Формула Ньютона-Лейбница: $\int_a^b f'(x)dx = f(b) - f(a)$

Пример: $\ln'(n) = \frac{1}{n} \Rightarrow \int_1^n \frac{1}{x}dx = \ln n - \ln 1 = \ln n$

2.8. Примеры по теме асимптотики

• Вложенные циклы for

```

1 #define forn(i, n) for (int i = 0; i < n; i++)
2 int counter = 0, n = 100;
3 forn(i, n)
4     forn(j, i)
5         forn(k, j)
6             forn(l, k)
7                 forn(m, l)
8                     counter++;
9 cout << counter << endl;

```

Чему равен counter? Во-первых, есть точный ответ: $\binom{n}{5} \approx \frac{n^5}{5!}$. Во-вторых, мы можем сходно посчитать число циклов и оценить ответ как $\mathcal{O}(n^5)$, правда константа $\frac{1}{120}$ важна, оценка через \mathcal{O} не даёт полное представление о времени работы.

• За сколько вычисляется n -е число Фибоначчи?

```

1 f[0] = f[1] = 1;
2 for (int i = 2; i < n; i++)
3     f[i] = f[i - 1] + f[i - 2];

```

Казалось бы за $\mathcal{O}(n)$. Но это в предположении, что “+” выполняется за $\mathcal{O}(1)$. На самом деле мы знаем, что $\log f_n = \Theta(n)$, т.е. складывать нужно числа длины $n \Rightarrow$ “+” выполняется за $\Theta(i)$, а n -е число Фибоначчи считается за $\Theta(n^2)$.

• Задача из теста про $a^2 + b^2 = N$

```

1 int b = sqrt(N);
2 for (int a = 1; a * a <= N; a++)
3     while (a * a + b * b >= N; b--)
4         ;
5     if (a * a + b * b == N)
6         cnt++;

```

Время работы $\Theta(N^{1/2})$, так как в сумме b уменьшится лишь $N^{1/2}$ раз. Здесь мы первый раз использовали так называемый “метод двух указателей”.

• Число делителей числа

```

1 vector<int> divisors[n + 1]; // все делители числа
2 for (int a = 1; a <= n; a++)
3     for (int b = a; b <= n; b += a)
4         divisors[b].push_back(a);

```

За сколько работает программа?

$$\sum_{a=1}^n \left\lceil \frac{n}{a} \right\rceil = \mathcal{O}(n) + \sum_{a=1}^n \frac{n}{a} = \mathcal{O}(n) + n \sum_{a=1}^n \frac{1}{a} \stackrel{2.7.5}{=} \mathcal{O}(n) + n \cdot \Theta\left(\int_1^n \frac{1}{x} dx\right) = \Theta(n \log n)$$

• Сумма гармонического ряда

Докажем более простым способом, что $\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$

$$1 + \lfloor \log_2 n \rfloor \geq \frac{1}{1} + \overbrace{\frac{1}{2} + \frac{1}{2}}^1 + \overbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}^1 + \overbrace{\frac{1}{8} + \dots}^{1 \dots} \geq \sum_{k=1}^n \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \dots \geq$$

$$\frac{1}{1} + \underbrace{\frac{1}{2}}_{1/2} + \underbrace{\frac{1}{4} + \frac{1}{4}}_{1/2} + \underbrace{\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}}_{1/2} + \dots \geq 1 + \frac{1}{2} \lfloor \log_2 n \rfloor \Rightarrow \sum_{k=1}^n \frac{1}{k} = \Theta(\log n)$$

2.9. Сравнение асимптотик

Def 2.9.1. Линейная сложность	$\mathcal{O}(n)$
Def 2.9.2. Квадратичная сложность	$\mathcal{O}(n^2)$
Def 2.9.3. Полиномиальная сложность	$\exists k > 0: \mathcal{O}(n^k)$
Def 2.9.4. Полилогарифм	$\exists k > 0: \mathcal{O}(\log^k n)$
Def 2.9.5. Экспоненциальная сложность	$\exists c > 0: \mathcal{O}(2^{cn})$

Теорема 2.9.6. $\forall x, y > 0, z > 1 \exists N \forall n > N: \log^x n < n^y < z^n$

Доказательство. Сперва докажем первую часть неравенства через вторую.

Пусть $\log n = k$, тогда $\log^x n < n^y \Leftrightarrow k^x < 2^{ky} = (2^y)^k = z^k \Leftarrow n^y < z^n$ ■

Докажем вторую часть исходного неравенства $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y} n \log z}$

Пусть $n' = \frac{1}{y} n \log z$, обозначим $C = 1/(\frac{1}{y} \log z)$, пусть $C \leq n'$ (возьмём достаточно большое n),

тогда $n^y < z^n \Leftrightarrow n < 2^{\frac{1}{y} n \log z} \Leftrightarrow C \cdot n' < 2^{n'} \Leftarrow (n')^2 < 2^{n'}$

Осталось доказать $n^2 < 2^n$. Докажем по индукции.

База: для любого значения из интервала $[10..20)$ верно,

так как $n^2 \in [100..400) < 2^n \in [1024..1048576)$.

Если n увеличить в два раза, то $n^2 \rightarrow 4 \cdot n^2$, а $2^n \rightarrow 2^{2n} = 2^n \cdot 2^n \geq 4 \cdot 2^n$ при $n \geq 2$.

Значит $\forall n \geq 2$ если для n верно, то и для $2n$ верно.

Переход: $[10..20) \rightarrow [20..40) \rightarrow [40..80) \rightarrow \dots$ ■

Следствие 2.9.7. $\forall x, y > 0, z > 1: \log^x n = \mathcal{O}(n^y), n^y = \mathcal{O}(z^n)$

Доказательство. Возьмём константу 1. ■

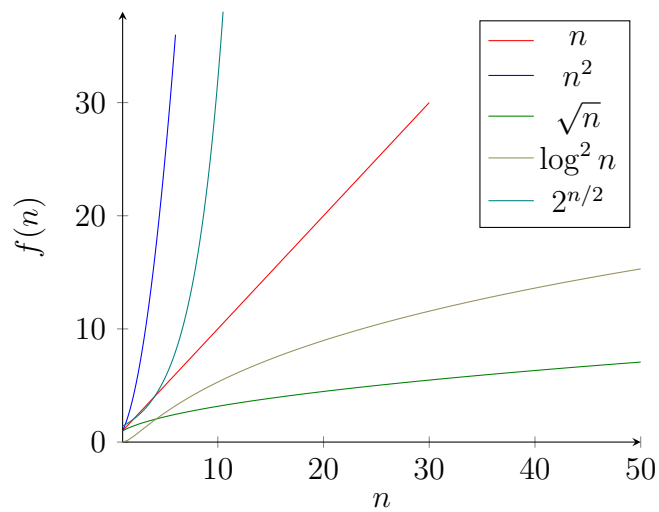
Следствие 2.9.8. $\forall x, y > 0, z > 1: \log^x n = o(n^y), n^y = o(z^n)$

Доказательство. Достаточно перейти к чуть меньшим y, z и воспользоваться теоремой.

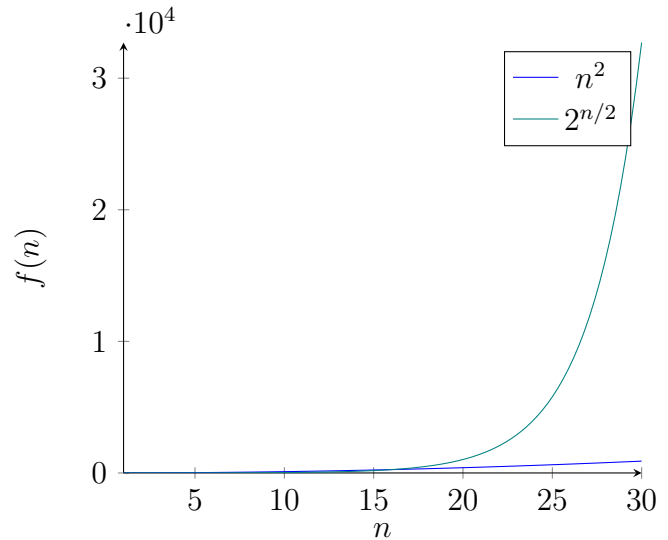
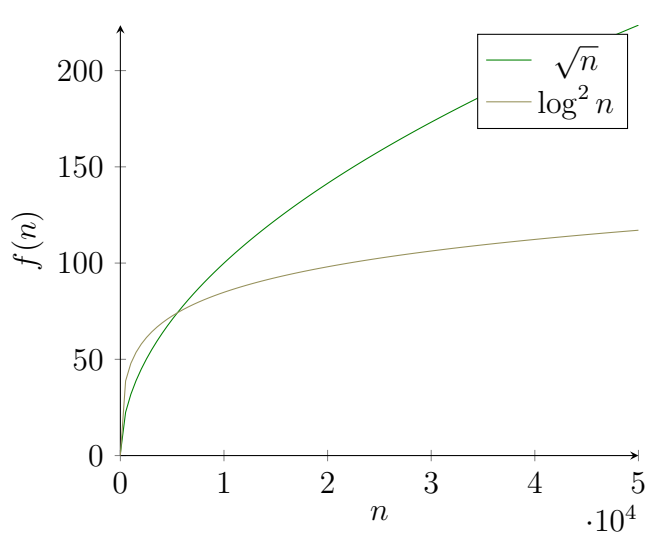
$\exists N \forall n \geq N \log^x n < n^{y-\varepsilon} = \frac{1}{n^\varepsilon} n^y, \frac{1}{n^\varepsilon} \xrightarrow{n \rightarrow \infty} 0 \Rightarrow \log^x n = o(n^y)$.

$\exists N \forall n \geq N n^y < (z - \varepsilon)^n = \frac{1}{(z/(z-\varepsilon))^n} z^n, \frac{1}{(z/(z-\varepsilon))^n} \xrightarrow{n \rightarrow \infty} 0 \Rightarrow n^y = o(z^n)$. ■

• Посмотрим как ведут себя функции на графике



Заметим, что $2^{n/2}$, n^2 и $\log^2 n$, \sqrt{n} на бесконечности ведут себя иначе:



Лекция #3: Структуры данных

11 сентября

3.1. C++

• Warnings

1. Сделайте, чтобы компилятор g++/clang отображал вам как можно больше warning-ов:

```
-Wall -Wextra -Wshadow
```

2. Пишите код, чтобы при компиляции не было warning-ов.

• Range check errors

Давайте рассмотрим стандартную багу: `int a[3]; a[3] = 7;`

В результате мы получаем *undefined behavior*. Код иногда падает по *runtime error*, иногда нет.

Чтобы такого не было, во-первых, используйте вектора, во-вторых, включите debug-режим.

```
1 #define _GLIBCXX_DEBUG // должна быть до всех #include
2 vector<int> a(3);
3 a[3] = 7; // 100% Runtime Error!
```

Для пользователей linux есть более профессиональное решение: **valgrind**.

• Struct (структуры)

```
1 struct Point {
2     int x, y;
3 };
4 Point p, q = {2, 3}, *t = new Point {2, 3};
5 p.x = 3;
```

• Pointers (указатели)

Рассмотрим указатель `int *a;`

`a` – указатель на адрес в памяти (по сути целое число, номер ячейки).

`*a` – значение, которое лежит по адресу.

```
1 int b = 3;
2 int *a = &b; // сохранили адрес b в переменную a типа int*
3 int c[10];
4 a = c; // указатель на первый элемент массива
5 *a = 7; // теперь c[0] == 7
6 Point *p = new Point {0, 0}; // выделили память под новый Point, указтель записали в p
7 (*p).x = 3; // записали значение в x
8 p->x = 3; // запись, эквивалентная предыдущей
```


3.2. Неасимптотические оптимизации

При написании программы, если хочется, чтобы она работала быстро, стоит обращать внимание не только на асимптотику, но и избегать использования некоторых операций, которые работают дольше, чем кажется.

1. Ввод и вывод данных. `cin/cout`, `scanf/printf`...
Используйте буфферизированный ввод/вывод через `fread/fwrite`.
2. Операции библиотеки `<math.h>`: `sqrt`, `cos`, `sin`, `atan` и т.д.
Эти операции раскладывают переданный аргумент в ряд, что происходит не за $\mathcal{O}(1)$.
3. Взятие числа по модулю, деление с остатком: `a / b`, `a % b`.
4. Доступ к памяти. Существует два способа прохода по массиву:
Random access: `for (i = 0; i < n; i++) sum += a[p[i]]`; здесь p – случайная перестановка
Sequential access: `for (i = 0; i < n; i++) sum += a[i]`;
5. Функции работы с памятью: `new`, `delete`. Тоже работают не за $\mathcal{O}(1)$.
6. Вызов функций. Пример, который при $n = 10^7$ работает секунду и использует ≥ 320 mb.

```
1 void go( int n ) {
2     if (n <= 0) return;
3     go(n - 1); // компилируйте с -O0, чтобы оптимизатор не раскрыл хвостовую рекурсию в цикл
4 }
```

Для оптимизации можно использовать `inline` – указание оптимизатору, что функцию следует не вызывать, а попытаться вставить в код.

• История про кеш

В нашем распоряжении есть примерно такие объёмы

1. Жёсткий диск. Самая медленная память, 1 терабайт.
2. Оперативная память. Средняя, 8 гигабайта.
3. Кеш L3. Быстрая, 4 мегабайта.
4. Кеш L1. Сверхбыстрая, 32 килобайта.

Отсюда вывод. Если у нас есть два алгоритма $\langle T_1, M_1 \rangle$ и $\langle T_2, M_2 \rangle$: $T_1 = T_2 = \mathcal{O}(n^2)$; $M_1 = \mathcal{O}(n^2)$; $M_2 = \Theta(n)$, то второй алгоритм будет работать быстрее для больших значений n , так как у первого будут постоянные промахи мимо кеша.

И ещё один. Если у нас есть два алгоритма $\langle T_1, M_1 \rangle$ и $\langle T_2, M_2 \rangle$: $T_1 = T_2 = \Theta(2^n)$; $M_1 = \Theta(2^n)$; $M_2 = \Theta(n^2)$, То первый в принципе не будет работать при $n \approx 40$, ему не хватит памяти. Второй же при больших $n \approx 40$ неспешно, за несколько часов, но отработает.

• Быстрые операции

`memcpy(a, b, n)` (скопировать n байт памяти), `strcmp(s, t)` (сравить строки).

Работают в 8 раз быстрее цикла `for` за счёт 128-битных SSE и 256-битных AVX регистров!

3.3. Частичные суммы

Дан массив $a[]$ длины n , нужно отвечать на большое число запросов $\text{get}(l, r)$ – посчитать сумму на отрезке $[l, r]$ массива $a[]$.

Наивное решение: на каждый запрос отвечать за $\Theta(r - l + 1) = \mathcal{O}(n)$.

Префиксные или частичные суммы:

```

1 void precalc() { // предподсчёт за  $\mathcal{O}(n)$ 
2     sum[0] = 0;
3     for (int i = 0; i < n; i++) sum[i + 1] = sum[i] + a[i]; //  $[0..i]$ 
4 }
5 int get( int l, int r ) { //  $[l..r]$ 
6     return sum[r + 1] - sum[l]; //  $[0..r] - [0..l)$ ,  $\mathcal{O}(1)$ 
7 }
```

3.4. Массив

Создать массив целых чисел на n элементов: `int a[n];`

Индексация начинается с 0, массивы имеют фиксированный размер. Функции:

1. `get(i)` – $a[i]$, обратиться к элементу массива с номером i , $\mathcal{O}(1)$
2. `set(i, x)` – $a[i] = x$, присвоить элементу под номером i значение x , $\mathcal{O}(1)$
3. `find(x)` – найти элемент со значением x , $\mathcal{O}(n)$
4. `add_begin(x)`, `add_end(x)` – добавить элемент в начало, в конец, $\mathcal{O}(n)$
5. `del_begin(x)`, `del_end(x)` – удалить элемент из начала, из конца, $\mathcal{O}(n)$

Последние команды работают долго т.к. нужно найти новый кусок памяти нужного размера, скопировать весь массив туда, удалить старый.

Другие названия для добавления: `insert`, `append`, `push`.

Другие названия для удаления: `remove`, `erase`, `pop`.

3.5. Двусвязный список

```

1 struct Node {
2     Node *prev, *next; // указатели на следующий и предыдущий элементы списка
3     int x;
4 };
5 struct List {
6     Node *head, *tail; // head, tail - фиктивные элементы
7 };
```

<code>get(i)</code> , <code>set(i, x)</code>	$\mathcal{O}(1)$
<code>find(x)</code>	$\mathcal{O}(n)$
<code>add_begin(x)</code> , <code>add_end(x)</code>	$\mathcal{O}(1)$
<code>del_begin()</code> , <code>del_end()</code>	$\mathcal{O}(1)$
<code>delete(Node*)</code>	$\mathcal{O}(1)$

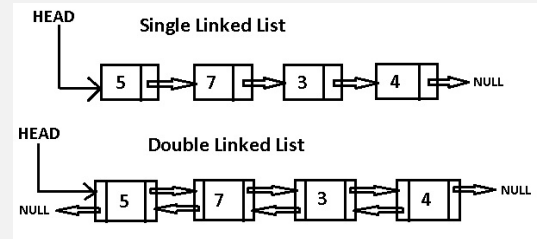
Указатель `tail` нужен, чтобы иметь возможность добавлять в конец, удалять из конца за $\mathcal{O}(1)$.

Ссылки `prev`, чтобы ходить по списку в обратном направлении, удалять из середины за $\mathcal{O}(1)$.

```

1 Node *find( List l, int x ) { // найти в списке за линию
2   for (Node *p = l.head->next; p != l.tail; p = p->next)
3     if (p->x == x)
4       return p;
5   return 0;
6 }
7 Node *erase( Node *v ) {
8   v->prev->next = v->next;
9   v->next->prev = v->prev;
10 }
11 Node *push_back( List &l, Node *v ) {
12   Node *p = new Node();
13   p->x = x, p->prev = l.tail->prev, p->next = l.tail;
14   p->prev->next = p, p->next->prev = p;
15 }
16 void makeEmpty( List &l ) { // создать новый пустой список
17   l.head = new Node(), l.tail = new Node();
18   l.head->next = l.tail, l.tail->prev = l.head;
19 }

```



3.6. Односвязный список

```

1 struct Node {
2   Node *next; // не храним ссылку назад, нельзя удалять из середины за O(1)
3   int x;
4 };
5 // 0 - пустой список
6 Node *head = 0; // не храним tail, нельзя добавлять в конец за O(1)
7 void push_front(Node* &head, int x) {
8   Node *p = new Node();
9   p->x = x, p->next = head, head = p;
10 }

```

3.7. Список на массиве

```

1 vector<Node> a; // массив всех Node-ов списка
2 struct {
3   int next, x;
4 };
5 int head = -1;
6 void push_front(int &head, int x) {
7   a.push_back(Node {head, x});
8   head = a.size() - 1;
9 }

```

Можно сделать свои указатели.

Тогда `next` – номер ячейки массива (указатель на ячейку массива).

3.8. Вектор (расширяющийся массив)

Обычный массив не удобен тем, что его размер фиксирован заранее и ограничен. Идея улучшения: выделим заранее *size* ячеек памяти, когда реальный размер массива n станет больше *size*, удвоим *size*, перевыделим память. Операции с вектором:

<code>get(i), set(i, x)</code>	$\mathcal{O}(1)$ (как и у массива)
<code>find(x)</code>	$\mathcal{O}(n)$ (как и у массива)
<code>push_back(x)</code>	$\Theta(1)$ (в среднем)
<code>pop_back()</code>	$\Theta(1)$ (в худшем)

```

1 int size, n, *a;
2 void push_back(int x) {
3     if (n == size) {
4         int *b = new int[2 * size];
5         copy(a, a + size, b);
6         a = b, size *= 2;
7     }
8     a[n++] = x;
9 }
10 void pop_back() { n--; }
```

Теорема 3.8.1. Среднее время работы одной операции $\mathcal{O}(1)$

Доказательство. Заметим, что перед удвоением размера $n \rightarrow 2n$ будет хотя бы $\frac{n}{2}$ операций `push_back`, значит среднее время работы последней всех `push_back` между двумя удвоениями, включая последнее удвоение $\mathcal{O}(1)$ ■

3.9. Стек, очередь, дек

Это названия интерфейсов (множеств функций, доступных пользователю)

Стек (stack)	<code>push_back</code> за $\mathcal{O}(1)$, <code>pop_back</code> за $\mathcal{O}(1)$. First In Last Out.
Очередь (queue)	<code>push_back</code> за $\mathcal{O}(1)$, <code>pop_front</code> за $\mathcal{O}(1)$. First In First Out.
Дек (deque)	все 4 операции добавления/удаления.

Реализовывать все три структуры можно, как на списке так и на векторе.

Деку нужен двусвязный список, очереди и стеку хватит односвязного.

Вектор у нас умеет удваиваться только при `push_back`. Что делать при `push_front`?

1. Можно удваиваться в другую сторону.
2. Можно использовать циклический вектор.

• Дек на циклическом векторе

```

deque:      { vector<int>a; int start, end; }, данные хранятся в [start, end)
sz():      { return a.size(); }
n():       { return end - start + (start <= end ? 0 : sz()); }
get(i):    { return a[(i + start) % sz()]; }
push_front(x): { start = (start - 1 + sz()) % sz(), a[start] = x; }
```

3.10. Очередь, стек и дек с минимумом

В стеке можно поддерживать минимум.

Для этого по сути нужно поддерживать два стека – стек данных и стек минимумов.

- **Стек с минимумом** – это два стека.

`push(x): a.push(x), m.push(min(m.back(), x))`

Здесь `m` – “частичные минимумы”, стек минимумов.

- **Очередь с минимумом через два стека**

Чтобы поддерживать минимум на очереди проще всего представить её, как два стека *a* и *b*.

```
1 Stack a, b;
2 void push(int x) { b.push(x); }
3 int pop() {
4     if (a.empty()) // стек a закончился, пора перенести элементы b в a
5         while (b.size())
6             a.push(b.pop());
7     return a.pop();
8 }
9 int getMin() { return min(a.getMin(), b.getMin()); }
```

- **Дек с минимумом через два стека**

TODO

Будет решено на практике.

Лекция #4: Структуры данных

18 сентября

4.1. Амортизационный анализ

Мы уже два раза оценивали время в среднем – для вектора и очереди с минимумом. Для более сложных случаев есть специальная система оценки “времени работы в среднем”, которую называют “амортизационным анализом”.

Пусть наша программа состоит из элементарных кусков (операций), i -й из которых работает t_i .

Def 4.1.1. t_i – *real time* (реальное время одной операции)

$m = \frac{\sum_i t_i}{n}$ – *average time* (среднее время)

$a_i = t_i + \Delta\varphi_i$ – *amortized time* (амортизированное время одной операции)

$\Delta\varphi_i = \varphi_{i+1} - \varphi_i$ – изменение функции φ , вызванное i -й операцией.

a_i – время, амортизированное функцией φ . Можно рассматривать любую φ .

• Пример: вектор.

Рассмотрим $\varphi = -size$ (размер вектора, взяли такой потенциал из головы).

1. Нет удвоения: $a_i = t_i + \Delta\varphi_i = 1 + 0 = \mathcal{O}(1)$

2. Есть удвоение: $a_i = t_i + \Delta\varphi_i = size + (\varphi_{i+1} - \varphi_i) = size - 2size + size = 0 = \mathcal{O}(1)$

Получили $a_i = \mathcal{O}(1)$, хочется сделать из этого вывод, что $m = \mathcal{O}(1)$

• Строгие рассуждения.

Lm 4.1.2. $\sum a_i = \sum t_i + (\varphi_{end} - \varphi_0)$

Доказательство. Сложили равенства $a_i = t_i + (\varphi_{i+1} - \varphi_i)$ ■

Теорема 4.1.3. $m = \mathcal{O}(\max a_i) + \mathcal{O}(|\frac{\varphi_{end} - \varphi_{start}}{n}|)$

Доказательство. В лемме делим равенство на n , $\sum a_i/n \leq \max a_i$, заменяем $\frac{\sum_i t_i}{n}$ на m ■

Следствие 4.1.4. Если $\varphi_0 = 0, \forall i \varphi_i \geq 0$, то $m = \mathcal{O}(\max a_i)$

• Пример: push, pop(k)

Пусть есть операции **push** за $\mathcal{O}(1)$ и **pop(k)** – достать сразу k элементов за $\Theta(k)$.

Докажем, что в среднем время любой операции $\mathcal{O}(1)$. Возьмём $\varphi = size$

push: $a_i = t_i + \Delta\varphi = 1 + 1 = \mathcal{O}(1)$

pop: $a_i = t_i + \Delta\varphi = k - k = \mathcal{O}(1)$

Также заметим, что $0 \leq \varphi \leq n$, поэтому $|\varphi_{end} - \varphi_{start}| \leq n$.

• Пример: $a^2 + b^2 = N$

```
1 int y = sqrt(n), cnt = 0;
2 for (int x = 0; x * x <= n; x++)
3     while (x * x + y * y > n) y--;
4     if (x * x + y * y == n) cnt++;
```

Одной операцией назовём итерацию внешнего цикла **for**.

Рассмотрим сперва корректный потенциал $\varphi = y$.

$$a_i = t_i + \Delta\varphi = (y_{old} - y_{new} + 1) + (y_{new} - y_{old}) = \mathcal{O}(1)$$

Также заметим, что $0 \leq \varphi \leq \sqrt{n}$, поэтому $|\varphi_{end} - \varphi_{start}| \leq \sqrt{n}$.

Теперь рассмотрим плохой потенциал $\bar{\varphi} = y^2$.

$$a_i = t_i + \Delta\bar{\varphi} = (y_{old} - y_{new} + 1) + (y_{new}^2 - y_{old}^2) = \mathcal{O}(1)$$

Но, при этом $\varphi_{start} = n, \varphi_{end} = 0$, поэтому нам по теореме не получится сделать вывод, что среднее время одной операции (итерации цикла `for`) равно $\mathcal{O}(1)$.

Теперь рассмотрим другой плохой потенциал $\tilde{\varphi} = 0$.

$$a_i = t_i + \Delta\tilde{\varphi} = (y_{old} - y_{new} + 1) = \mathcal{O}(\sqrt{n})$$

• Монетки

Докажем ещё одним способом, что вектор работает в среднем за $\mathcal{O}(1)$.

Когда мы делаем `push_back` без удвоения памяти, накопим 2 монетки.

Когда мы делаем `push_back` с удвоением $size \rightarrow 2size$, это занимает $size$ времени, но мы можем заплатить за это, потратив $size$ накопленных монеток. Число денег никогда не будет меньше нуля, так как до удвоения было хотя бы $\frac{size}{2}$ операций “`push_back` без удвоения”.

Эта идея равносильна идее про потенциалы. Мы неявно определяем функцию φ через её $\Delta\varphi$. φ – количество накопленных и ещё не потраченных монеток. $\Delta\varphi$ = соответственно +2 и $-size$.

4.2. Разбор арифметических выражений

Разбор выражений с числами, скобками, операциями.

Предположим, все операции левоассоциативны (вычисляются слева направо).

```

1 stack<int> value; // уже посчитанные значения
2 stack<char> op; // ещё не выполненные операции
3 void make() { // выполнить правую операцию
4     int b = value.top(); value.pop();
5     int a = value.top(); value.pop();
6     char o = op.top(); op.pop();
7     value.push(a o b); // да, не скомпилился, но смысл такой
8 }
9 int eval(string s) { // s без пробелов
10    s = '(' + s + ')'; // при выполнении последней ), выражение вычислится
11    for (char c : s)
12        if ('0' <= c && c <= '9') value.push(c - '0');
13        else if (c == '(') op.push(c);
14        else if (c == ')') {
15            while (op.top() != '(') make();
16            op.pop();
17        } else {
18            while (op.size() && priority(op.top()) >= priority(c)) make();
19            op.push(c);
20        }
21    return value.top();
22 }
```

Теорема 4.2.1. Время разбора выражения s со стеком равно $\Theta(|s|)$

Доказательство. В функции `eval` число вызовов `push` не больше $|s|$. Операция `make` уменьшает размер стеков, поэтому число вызовов `make` не больше числа операций `push` в функции `eval`. ■

4.3. Бинпоиск

4.3.1. Обыкновенный

Дан отсортированный массив. Сортировать мы пока умеем только так:

```
int a[n]; sort(a, a + n);
vector<int> a(n); sort(a.begin(), a.end());
```

Сейчас мы научимся за $\mathcal{O}(\log n)$ искать в этом массиве элемент x

```
1 bool find( int l, int r, int x ) { // [l,r]
2   while (l <= r) {
3     int m = (l + r) / 2;
4     if (a[m] == x) return m;
5     if (a[m] < x) l = m + 1;
6     else r = m - 1;
7   }
8   return 0;
9 }
```

Lm 4.3.1. Время работы $\mathcal{O}(\log n)$

Доказательство. Каждый раз мы уменьшаем длину отрезка $[l, r]$ как минимум в 2 раза. ■

• **Lowerbound.** Можно искать более сложную величину $\min i: a_i \geq x$.

```
1 int lower_bound( int l, int r, int x ) { // [l,r]
2   while (l < r) {
3     int m = (l + r) / 2;
4     if (a[m] < x) l = m + 1;
5     else r = m;
6   }
7   return l;
8 }
```

Если все элементы $[l, r)$ меньше x , `lower_bound` вернёт r . Время работы также $\mathcal{O}(\log n)$.

Заметим, что этот бинпоиск строго сильнее, функцию `find` теперь можно реализовать так:

```
return a[lower_bound(l, r, x)] == x;
```

В языке C++ есть стандартные функции

```
1 int i = lower_bound(a, a + n, x) - a; // min i: a[i] >= x
2 int i = upper_bound(a, a + n, x) - a; // min i: a[i] > x
```

Через них легко найти $[\max i: a_i \leq x] = \text{upper_bound} - 1$ и $[\max i: a_i < x] = \text{lower_bound} - 1$.

4.3.2. По предикату

Можно написать ещё более общий бинпоиск, при этом сделать код более простым. Пусть есть функция булева функция (предикат) f , которая до некоторой позиции i имеет значение 0, а, начиная с i , имеет значение 1.

Тогда мы бинпоиском можем найти такие $l + 1 = r$, что $f(l) = 0, f(r) = 1$

```
1 void find_predicate( int &l, int &r ) { // f(l) = 0, f(r) = 1
2   while (r - l > 1) {
3     int m = (l + r) / 2;
```



```

4   (f(m) ? r : l) = m;
5   }
6 }

```

Как это использовать для решения задачи `lower_bound`?

```

1 bool f( int i ) { return a[i] >= x; }
2 int l = -1, r = n;
3 find_predicate(l, r); // f() будет вызываться только для элементов от l+1 до r-1
4 return r; // f(r) = 1, f(r-1) = 0

```

4.3.3. Вещественный, корни многочлена

Дан многочлен P нечётной степени со старшим коэффициентом 1. У него есть вещественный корень и мы можем его найти бинарным поиском с любой наперёд заданной точностью ε .

Сперва нужно найти точки l, r : $P(l) < 0, P(r) > 0$.

```

1 for (l = -1; P(l) >= 0; l *= 2) ;
2 for (r = +1; P(r) <= 0; r *= 2) ;

```

Теперь собственно поиск корня:

```

1 while (r - l > ε) {
2     double m = (l + r) / 2;
3     (P(m) < 0 ? l : r) = m;
4 }

```

Внешний цикл может быть бесконечным из-за погрешности.

Пример: $l = 10^9, r = 10^9 + 10^{-6}, \varepsilon = 10^{-9}$. Чтобы он точно завершился, посчитаем, сколько мы хотим итераций: $k = \log_2 \frac{r-l}{\varepsilon}$, и сделаем ровно k итераций: `for (int i = 0; i < k; i++)`.

Поиск всех вещественных корней многочлена степени n будет через месяц на практике.

4.4. Два указателя и операции над множествами

Множества можно хранить в виде отсортированных массивов. Наличие элемента в множестве тогда можно проверять за $\mathcal{O}(\log n)$, а элементы перебирать за линейное время. Также за линейное время, зная A и B , методом “двух указателей” можно найти $A \cap B, A \cup B, A \setminus B$. Также можно искать объединение мультимножеств. В языке C++ это операции `set_intersection`, `set_union`, `set_difference`, `merge`. Все они имеют синтаксис `k = merge(a, a+n, b, b+m, c)` - c , где c - указатель на область результата, память выделить должны вы сами, а k - количество элементов в ответе. Пример применения “двух указателей” для поиска пересечения.

Вариант #1, for:

```

1 B[|B|] = +∞; // барьерный элемент
2 for (int k = 0, j = 0, i = 0; i < |A|; i++) {
3     while (B[j] < A[i]) j++;
4     if (B[j] == A[i]) C[k++] = A[i];
5 }

```

Вариант #2, while:

```

1 int i = 0, j = 0;
2 while (i < |A| && j < |B|)
3     if (A[i] == B[j]) C[k++] = A[i++], j++;
4     else (A[i] < B[j] ? i : j)++;

```

4.5. Хеш-таблица

Структура данных, умеющая делать операции **add**, **del**, **find** за рандомизированное $\mathcal{O}(1)$. Принципиально лучше вектора, который умеет делать **find** только за $\mathcal{O}(n)$.

4.5.1. На списках

```
1 list<int> h[N]; // собственно хеш-таблица
2 void add( int x ) { h[x % N].push_back(x); } //  $\mathcal{O}(1)$  в худшем
3 auto find( int x ) { return find(h[x % N].begin(), h[x % N].end(), x); }
4 // find работает за длину списка
5 void erase( int x ) { h[x % N].erase(find(x)); } // работает за find +  $\mathcal{O}(1)$ 
```

Вместо **list** можно использовать любую структуру данных, **vector**, или даже хеш-таблицу.

Если в хеш-таблице живёт n элементов и они равномерно распределены по спискам, в каждом списке $\frac{n}{N}$ элементов \Rightarrow при $n \leq N$ и равномерном распределении элементов, все операции работают за $\mathcal{O}(1)$. Как сделать распределение равномерным? Подобрать хорошую хеш-функцию!

Утверждение 4.5.1. Если N простое, то хеш-функция $x \rightarrow x \% N$ достаточно хорошая.

Без доказательства.

Если добавлять в хеш-таблицу новые элементы, со временем n станет больше N .

В этот момент нужно перевыделить память $N \rightarrow 2N$ и передобавить все элементы на новое место. Возьмём $\varphi = -N \Rightarrow$ амортизированное время удвоения $\mathcal{O}(1)$.

4.5.2. С открытой адресацией

Реализуется на одном циклическом массиве. Хеш-функция используется, чтобы получить начальное значение ячейки. Далее двигаемся вправо, пока не найдём ячейку, в которой живёт наш элемент или свободную ячейку, куда можно его поселить.

```
1 unsigned h[N]; // собственно хеш-таблица
2 int getIndex( unsigned x ) { // поиск индекса по элементу
3     int i = x % N; // используем хеш-функцию
4     while (h[i] && h[i] != x) // x != 0, ноль обозначает свободную ячейку
5         if (++i == N) // массив циклический
6             i = 0;
7     return i;
8 }
```

1. **Добавление:** `h[getIndex(x)] = x;`
2. **Удаление:** `h[getIndex(x)] = -1;` нужно потребовать `x != -1`, ячейка не становится свободной.
3. **Поиск:** `return h[getIndex(x)] != 0;`

Lm 4.5.2. Если в хеш-таблице с открытой адресацией размера N занято αN ячеек, $\alpha < 1$, матожидание время работы `getIndex` не более $\frac{1}{1-\alpha}$.

Доказательство. Худший случай – x отсутствует в хеш-таблице. Без доказательства предположим, что свободные ячейки при хорошей хеш-функции расположены равномерно. Тогда на каждой итерации цикла `while` вероятность “не остановки” равна α . Вероятность того, что мы не остановимся и после k шагов равна α^k . Значит время работы равно $1 + \alpha + \alpha^2 + \alpha^3 + \dots = \frac{1}{1-\alpha}$. ■

• Переполнение хеш-таблицы

При слишком малом α операции с хеш-таблицей начинают работать долго.

При $\alpha = 1$ (нет свободных ячеек), `getIndex` будет бесконечно искать свободную. Что делать?

При $\alpha > \frac{2}{3}$ удваивать размер и за линейное время передобавлять все элементы в новую таблицу.

При копировании, конечно, пропустим все -1 (уже удалённые ячейки) \Rightarrow удалённые ячейки занимают лишнюю память ровно до ближайшего перевыделения памяти.

4.5.3. C++

`unordered_set<int> h;` – хеш-таблица, хранящая множество `int`-ов.

Использование:

1. `unordered_set<int> h(N);` выделить заранее память под N ячеек
2. `h.count(x);` проверить наличие x
3. `h.insert(x);` добавить x , если уже был, ничего не происходит
4. `h.erase(x);` удалить x , если его не было, ничего не происходит

`unordered_map<int, int> h;` – хеш-таблица, хранящая `pair<int, int>`, пары `int`-ов.

Использование:

1. `unordered_map<int, int> h(N);` выделить заранее память под N ячеек
2. `h[i] = x;` i -й ячейкой можно пользоваться, как обычным массивом
3. `h.count(i);` есть ли пара с первой половиной i (ключ i)
4. `h.erase(i);` удалить пару с первой половиной i (ключ i)

Относиться к `unordered_map` можно, как к обычному массиву с произвольными индексами.

В теории эта структура называется “ассоциативный массив”: каждому ключу i в соответствие ставится его значение $h[i]$.

Лекция #5: Структуры данных

25 сентября

5.1. Избавляемся от амортизации

Серьёзный минус вектора – амортизированное время работы. Сейчас мы модифицируем структуру данных, она начнёт чуть дольше работать, использовать чуть больше памяти, но время одной операции в худшем будет $\mathcal{O}(1)$.

5.1.1. Вектор (решаем проблему, когда случится)

В тот `push_back`, когда старый вектор `a` переполнился, выделим память под новый вектор `b`, новый элемент положим в `b`, копировать `a` пока не будем. Сохраним `pos = |a|`. Инвариант: первые `pos` элементов лежат в `a`, все следующие в `b`. Каждый `push_back` будем копировать по одному элементу.

```
1 int *a, *b; // выделенные области памяти
2 int pos; // разделитель скопированной и не скопированной частей
3 int n, size; // количество элементов; выделенная память
4 void push_back(int x) {
5     if (pos > 0) b[pos] = a[pos], pos--;
6     if (n == size) {
7         delete [] a; // мы его уже скопировали, он больше не нужен
8         a = b;
9         n = size, size *= 2;
10        pos = n, b = new int[size];
11    }
12    b[n++] = x;
13 }
```

Как мы знаем, `new` работает за $\mathcal{O}(\log n)$, это нас устроит.

Тем не менее в этом месте тоже можно получить $\mathcal{O}(1)$.

Lm 5.1.1. К моменту `n == size` вектор `a` целиком скопирован в `b`.

Доказательство. У нас было как минимум n операций `push_back`, каждая уменьшала `pos`. ■

Операция обращения к i -му элементу обращается теперь к $(i < pos ? a : b)$.

Время на копировании не увеличилось. Время обращения к i -му элементу чуть увеличилось (лишний `if`). Памяти в среднем теперь нужно в 1.5 раз больше, т.к. мы в каждый момент храним и старую, и новую версию вектора.

5.1.2. Вектор (решаем проблему заранее)

Сделаем так, чтобы время обращения к i -му элементу не изменилось.

Мы начнём копировать заранее, в момент `size = 2n`, когда вектор находится в нормальном состоянии. Нужно к моменту очередного переполнения получить копию вектора в память большего размера. За n `push_back`-ов должны успеть скопировать все `size = 2n` элементов. Поэтому будем копировать по 2 элемента. Когда в такой вектор мы записываем новые значения (`a[i]=x`), нам нужно записывать в обе версии – и старую, и новую.

5.1.3. Хеш-таблица

Хеш-таблица – ещё одна структура данных, которая при переполнении удваивается. К ней можно применить оба описанных подхода. Применим первый. Чтобы это сделать, достаточно научиться перебирать все элементы хеш-таблицы и добавлять их по одному в новую хеш-таблицу.

(а) Можно кроме хеш-таблицы дополнительно хранить “список добавленных элементов”.

(б) Можно пользоваться тем, что число ячеек не более чем в два раза больше числа элементов, поэтому будем перебирать ячейки, а из них выбирать не пустые. Новые элементы, конечно, мы будем добавлять только в новую хеш-таблицу.

5.1.4. Очередь с минимумом через два стека

Напомним, что есть очередь с минимумом.

```

1 Stack a, b;
2 void push(int x) { b.push(x); }
3 int pop() {
4     if (a.empty()) // стек a закончился, пора перенести элементы b в a
5         while (b.size())
6             a.push(b.pop());
7     return a.pop();
8 }
```

Воспользуемся вторым подходом “решаем проблему заранее”. К моменту, когда стек *a* опустеет, у нас должна быть уже готова перевёрнутая версия *b*. Вот общий шаблон кода.

```

1 Stack a, b, a1, b1;
2 void push(int x) {
3     b1.push(x); // кидаем не в b, а в b1, копию b
4     STEP; // сделать несколько шагов копирования
5 }
6 int pop() {
7     if (копирование завершено)
8         a = a1, b = b1, начать новое копирование;
9     STEP; // сделать несколько шагов копирования
10    return a.pop();
11 }
```

Почему нам вообще нужно копировать внутри `push`? Если мы делаем сперва 10^6 `push`, затем 10^6 `pop`, к моменту всех этих `pop` у нас уже должен быть подготовлен длинный стек *a*. Если в течение `push` мы его не подготовили, его взять неоткуда.

При копировании мы хотим построить новый стек *a1* по старым *a* и *b* следующим образом (`STEP` сделает несколько шагов как раз этого кода):

```

1 while (b.size()) a1.push(b.pop());
2 for (int i = 0; i < a.size(); i++) a1.push(a[i]);
```

Заметим, что `a.size()` будет меняться при вызовах `a.pop_back()`. `for` проходит элементы *a* снизу вверх. Так можно делать, если стек *a* реализован через вектор без амортизации. Из кода видно, что копирование состоит из $|a| + |b|$ шагов. Будем поддерживать инвариант, что до начала копирования $|a| \geq |b|$. В каждом `pop` будем делать 1 шаг копирования, в каждом `push` также 1 шаг. Проверка инварианта после серии `push`: за *k* пушей мы сделали $\geq k$ копирований, поэтому $|a_1| \geq |b_1|$. Проверка корректности `pop`: после первых $|b|$ операций `pop` все элементы *b* уже скопировались, далее мы докопируем часть *a*, которая не подверглась `pop_back`-ам.

5.2. Бинарная куча

Рассмотрим массив $a[1..n]$. Его элементы образуют бинарное дерево с корнем в 1. Дети i – вершины $2i$, $2i + 1$. Отец i – вершина $\lfloor \frac{i}{2} \rfloor$.

Def 5.2.1. *Бинарная куча – массив, индексы которого образуют описанное выше дерево, в котором верно основное свойство кучи: для каждой вершины i значение $a[i]$ является минимумом в поддереве i .*

Lm 5.2.2. Высота кучи равна $\lfloor \log_2 n \rfloor$

Доказательство. Высота равна длине пути от n до корня.

Заметим, что для всех чисел от 2^k до $2^{k+1} - 1$ длина пути в точности k . ■

• Интерфейс

Бинарная куча за $\mathcal{O}(\log n)$ умеет делать следующие операции.

1. `GetMin()`. Нахождение минимального элемента.
2. `Add(x)`. Добавление элемента.
3. `ExtractMin()`. Извлечение (удаление) минимума.

Если для элементов хранятся “обратные указатели”, позволяющие за $\mathcal{O}(1)$ переходить от элемента к ячейке кучи, содержащей элемент, то куча также за $\mathcal{O}(\log n)$ умеет:

4. `DecreaseKey(x, y)`. Уменьшить значение ключа x до y .
5. `Del(x)`. Удалить из кучи x .

5.2.1. GetMin, Add, ExtractMin

Реализуем сперва три простые операции.

Наша куча: `int n, *a;`. Память выделена, её достаточно.

```

1 void Init()      { n = 0; }
2 int GetMin()     { return a[1]; }
3 void Add(int x)  { a[++n] = x, siftUp(n); }
4 void ExtractMin() { swap(a[1], a[n--]), siftDown(1); }
5 // DelMin перед удалением сохранил минимум в a[n]
```

Здесь `siftUp` – проталкивание элемента вверх, а `siftDown` – проталкивание элемента вниз. Обе процедуры считают, что дерево обладает свойством кучи везде, кроме указанного элемента.

```

1 void siftUp(int i) {
2     while (i > 1 && a[i / 2] > a[i]) // пока мы не корень и отец нас больше
3         swap(a[i], a[i / 2]), i /= 2;
4 }
5 void siftDown(int i) {
6     while (1) {
7         int l = 2 * i;
8         if (l + 1 <= n && a[l + 1] < a[l]) l++; // выбрать меньшего из детей
9         if (!(l <= n && a[l] < a[i])) break; // если все дети не меньше нас, это конец
10        swap(a[l], a[i]), i = l; // перейти в ребёнка
11    }
12 }
```

Lm 5.2.3. Обе процедуры корректны

Доказательство. По индукции на примере `siftUp`. В каждый момент времени верно, что поддерево i – корректная куча. Когда мы выйдем из `while`, у i нет проблем с отцом, поэтому вся куча корректна из предположения “корректно было всё кроме i ”. ■

Lm 5.2.4. Обе процедуры работают за $\mathcal{O}(\log n)$

Доказательство. Они работают за высоту кучи, которая по 5.2.2 равна $\mathcal{O}(\log n)$. ■

5.2.2. Обратные ссылки и DecreaseKey

Давайте предположим, что у нас есть массив значений: `vector<int> value`.

В куче будем хранить индексы этого массива. Тогда все сравнения `a[i] < a[j]` следует заменить на сравнения через `value`: `value[a[i]] < value[a[j]]`. Чтобы добавить элемент, теперь нужно сперва добавить его в конец `value`: `value.push_back(x)`, а затем сделать добавление в кучу `Add(value.size() - 1)`. Хранение индексов позволяет нам для каждого i помнить позицию в куче `pos[i]: a[pos[i]] == i`. Значения `pos[]` нужно пересчитывать каждый раз, когда мы меняем значения `a[]`. Как теперь удалить произвольный элемент с индексом i ?

```
1 void Del(int i) {
2     i = pos[i];
3     h[i] = h[n--], pos[h[i]] = i; // не забыли обновить pos
4     siftUp(i), siftDown(i); // новый элемент может быть и меньше, и больше
5 }
```

Процедура `DecreaseKey(i)` делается похоже: перешли к `pos[i]`, сделали `siftUp`.

Lm 5.2.5. `Del` и `DecreaseKey` корректны и работают за $\mathcal{O}(\log n)$

Доказательство. Следует из корректности и времени работы `siftUp`, `siftDown` ■

5.2.3. Build, HeapSort

```
1 void Build( int n, int *a ) {
2     for (int i = n; i >= 1; i--)
3         siftDown(i);
4 }
```

Lm 5.2.6. Функция `Build` построит корректную бинарную кучу.

Доказательство. Когда мы проталкиваем i , по индукции слева и справа уже корректные бинарные кучи. По корректности операции `sift_down` после проталкивания i , поддерево i является корректной бинарной кучей. ■

Lm 5.2.7. Время работы функции `Build` $\Theta(n)$

Доказательство. Пусть $n = 2^k - 1$, тогда наша куча – полное бинарное дерево. На самом последнем (нижнем) уровне будет 2^{k-1} элементов, на предпоследнем 2^{k-2} элементов и т.д. `sift_down(i)` работает за $\mathcal{O}(\text{глубины поддерева } i)$, поэтому суммарное

время работы $\sum_{i=1}^k 2^{k-i} i = 2^k \sum_{i=1}^k \frac{i}{2^i} \stackrel{(*)}{=} 2^k \cdot \Theta(1) = \Theta(n)$. (*) доказано на практике. ■


```

1 void HeapSort() {
2     Build(n, a); // строим очередь с максимумом,  $O(n)$ 
3     for(i, n) DelMax(); // максимум окажется в конце и т.д.,  $O(n \log n)$ 
4 }

```

Lm 5.2.8. Функция `HeapSort` работает за $O(n \log n)$, использует $O(1)$ дополнительной памяти.

Доказательство. Важно, что функция `Build` не копирует массив, строит кучу прямо в `a`. ■

5.3. Аллокация памяти

Задача: реализовать две функции

1. `int new(int x)` выделяет `x` байт, возвращает адрес первой свободной ячейки
2. `void delete(int addr)` освобождает память, по адресу `addr`, которую когда-то вернул `new`

В общем случае задача сложная. Сперва рассмотрим популярное решение более простой задачи.

5.3.1. Стек

Разрешим освобождать не любую область памяти, а последнюю выделенную.

```

1 int pos = 0; // указатель на первую свободную ячейку
2 int new( unsigned x ) { // push(x)
3     pos += x;
4     assert(pos <= MAX_MEM); // проверить, что памяти всё ещё хватает
5     return pos - x;
6 }
7 void delete( unsigned x ) { // pop
8     pos = x; // очищать можно только последнюю выделенную
9 }

```

В C++ при вызове функции, создании локальных переменных используется ровно такая же модель аллокации памяти, называется также – “стек”. Иногда имеет смысл реализовать свой стек-аллокатор и перегрузить глобальный `operator new`, так как стандартные STL-контейнеры `vector`, `set` внутри много раз обращаются к стандартному медленному `operator new`.

5.3.2. Список

Ещё один частный простой случай `x = CONST`, все выделяемые ячейки одного размера. Пусть наше адресуемое пространство 32-битное, то есть, $\text{MAX_MEM} \leq 2^{32}$. Тогда давайте исходные `MAX_MEM` байт памяти разобьём на 4 байта `head` и на $k = \lfloor \frac{\text{MAX_MEM}-4}{\max(x,4)} \rfloor$ ячеек по $\max(x, 4)$ байт. Каждая из k ячеек или свободная, тогда она – “указатель на следующую свободную”, или занята, тогда она – “ x байт полезной информации”. `head` – начало списка свободных ячеек, первая свободная. Изначально все ячейки свободны и объединены в список.

```

1 char mem[MAX_MEM]; // наша память
2 unsigned new() { // вернёт целое число - адрес в нашем 32-битном пространстве mem
3     unsigned res = head;
4     head = *(unsigned*)(mem + head); // взяли 4-байтовое число по адресу head
5     return res;
6 }
7 void delete( unsigned x ) {
8     *(unsigned*)(mem + x) = head; // записали в первые 4 байта ячейки число x
9     head = x;
10 }

```


5.3.3. Куча и хеш-таблица

MAX_МЕМ байт памяти разобьются на подряд идущие куски свободного пространства и подряд идущие куски занятого пространства. Давайте все куски свободного пространства хранить в куче с максимумом в корне, сравнивающей куски по размеру. Кроме того будем для каждого свободного или занятого куска $[l..r]$ хранить $\text{pair}[l] = r$, $\text{pair}[r] = l$, pair – хеш-таблица.

- **Операция new(x).**

Если в корне кучи максимум меньше x , память не выделяется.

Иначе память выделяется за $\mathcal{O}(1) + \langle \text{время просеивания вниз в куче} \rangle = \mathcal{O}(\log n)$.

- **Операция delete.**

Нужно объединить несколько кусков свободной памяти в один. Хеш-таблица поможет.

Куче нужно уметь удалять из середины, для этого поддерживаем обратные ссылки.

- **Хеш-таблица.** Все обновления хеш-таблицы происходят за $\mathcal{O}(1)$.

Откуда брать память под кучу и хеш-таблицу? У себя же =). Рекурсивный вызов.

5.3.4. Куча

TODO

5.4. Пополняемые структуры

Все описанные в этом разделе идеи применимы не ко всем структурам данных. Тем не менее к любой структуре любую из описанных идей можно *попробовать* применить.

5.4.1. Ничего → Удаление

Вид “ленивого удаления”. Пример: куча. Есть операция `DelMin`, хотим операцию удаления произвольного элемента, ничего не делая. Будем хранить две кучи – добавленные элементы и удалённые элементы.

```

1 Heap a, b;
2 void Add(int x) { a.add(x); }
3 void Del(int x) { b.add(x); }
4 int DelMin() {
5     while (b.size() && a.min() == b.min())
6         a.delMin(), b.delMin(); // пропускаем уже удалённые элементы
7     return a.delMin();
8 }

```

Асимптотика не ухудшилась.

Время работы `DelMin` теперь амортизированное, в худшем случае $\Theta(n)$.

5.4.2. Поиск → Удаление

Вид “ленивого удаления”. Таким приёмом мы уже пользовались при удалении из хеш-таблицы с открытой адресацией. Идея: у нас есть операция `Find`, отлично, найдём элемент, пометим его, как удалённый. Удалять прямо сейчас не будем.

5.4.3. Add → Merge

Merge (слияние) – операция, получающая на вход две структуры данных, на выход даёт одну, равную их объединению. Старые структуры объявляются невалидными.

Пример #1. Merge двух сортированных массивов.

Пример #2. Merge двух куч. Сейчас мы научимся делать его быстро.

• **Идея.** У нас есть операция добавления одного элемента, переберём все элементы меньшей структуры данных и добавим их в большую.

```

1 Heap Merge(Heap a, Heap b) {
2     if (a.size < b.size) swap(a, b);
3     for (int x : b) a.Add(x);
4     return a;
5 }

```

Lm 5.4.1. Если мы начинаем с \emptyset и делаем N произвольных операций из множества $\{\text{Add}, \text{Merge}\}$, функция `Add` вызовется не более $N \log_2 N$ раз.

Доказательство. Посмотрим на код и заметим, что $|a| + |b| \geq 2|b|$, поэтому для каждого x , переданного `Add` верно, что “размер структуры, в которой живёт x , хотя бы удвоился” $\Rightarrow \forall x$ количество операций `Add`(x) не более $\log_2 N \Rightarrow$ суммарное число всех `Add` не более $N \log_2 N$. ■

5.4.4. Build \rightarrow Add

• Решение #1. Корневая.

Сперва разберём задачу из домашнего задания и поймём, что мы умеем делать операцию Add амортизированно за $\mathcal{O}(\sqrt{n})$.

Пример: сортированный массив.

Build = sort = $\mathcal{O}(n \log n)$. Rebuild = merge = $\mathcal{O}(n)$. Get = бинпоиск = $\mathcal{O}(\log n)$.

Структура данных: храним сортированный массив и новых k элементов, поддерживаем $k \leq \sqrt{\text{Rebuild}}$. Когда нам нужно добавить новый элемент, сортированный массив мы не трогаем, добавляем в его в пачку из k элементов.

Get: вызвать бинпоиск для сортированного массива, перебрать все $k \leq \sqrt{\text{Rebuild}}$ элементов.

Add: добавить элемент, $k++$, если k стало больше $\sqrt{\text{Rebuild}}$, вызвать Rebuild.

Среднее время работы Add равно $\sqrt{\text{Rebuild}} = \sqrt{n}$, суммарное время n операций Get/Add равно $\mathcal{O}(n\sqrt{\text{Rebuild}}) = \mathcal{O}(n\sqrt{n})$.

• Решение #2. Пополняемые структуры.

Пусть у нас есть структура S с интерфейсом S.Build, S.Get, S.AllElements. У любого числа N есть единственное представление в двоичной системе счисления $a_1 a_2 \dots a_k$. Для хранения $N = 2^{a_1} + 2^{a_2} + \dots + 2^{a_k}$ элементов будем хранить k структур S из $2^{a_1}, 2^{a_2}, \dots, 2^{a_k}$ элементов. $k \leq \log_2 n$. Новый Get работает за $k \cdot \text{S.Get}$, обращается к каждой из k частей. Сделаем Add(x). Для этого добавим ещё одну структуру из 1 элемента. Теперь сделаем так, чтобы не было структур одинакового размера.

```

1 for (i = 1; есть две структуры размера i; i *= 2)
2   Добавим S.Build(A.AllElements + B.AllElements). // A, B - те самые две структуры
3   Удалим две старые структуры

```

Заметим, что по сути мы добавляли к числу N единицу в двоичной системе счисления.

Lm 5.4.2. Пусть мы начали с пустой структуры, было n вызовов Add и k вызовов Build(a_1), Build(a_2), ..., Build(a_k). Тогда $\sum_{i=1}^k a_i \leq n \log_2 n$

Доказательство. Когда элемент проходит через Build размер структуры, в которой он живёт, удваивается. Поэтому каждый x пройдёт через Build не более $\log_2 n$ раз. ■

Lm 5.4.3. $\forall k \geq 1, a_i > 0: (\sum a_i)^k \geq \sum a_i^k$ (без доказательства)

Lm 5.4.4. Суммарное время обработки n запросов не более Build($n \log_2 n$)

Применение данной идеи для сортированного массива будем называть “пополняемый массив”.

5.4.5. Build \rightarrow Add, Del

Научим “пополняемый массив обрабатывать запросы”.

1. Count(l, r) – посчитать число $x: l \leq x \leq r$
2. Add(x) – добавить новый элемент
3. Del(x) – удалить ранее добавленный элемент

Для этого будем хранить два “пополняемых массива” – добавленные элементы, удалённый элемент. Когда нас просят сделать Count, возвращаем разность Count-ов за $\mathcal{O}(\log^2 n)$. Add и Del

работают амортизированно за $\mathcal{O}(\log n)$, так как вместо `Build`, который должен делать `sort`, мы вызовем `merge` двух отсортированных массивов за $\mathcal{O}(n)$.

5.5. Два указателя и алгоритм Мо

• **Задача:** дан массив длины n и m запросов вида

“количество различных чисел на отрезке $[l_i, r_i]$ ”.

Если $l_i \leq l_{i+1}, r_i \leq r_{i+1}$ – это обычный метод двух указателей с хеш-таблицей внутри. Решение работает за $\mathcal{O}(n + m)$ операций с хеш-таблицей. Такую идею можно применить и к другим типам запросов. Для этого достаточно, зная ответ и поддреживая некую структуру данных, для отрезка $[l, r]$ научиться быстро делать операции `l++`, `r++`.

Если же l_i и r_i произвольны, то есть решение за $\mathcal{O}(n\sqrt{m})$, что, конечно, хуже $\mathcal{O}(n + m)$, но гораздо лучше обычного $\mathcal{O}(nm)$.

• Алгоритм Мо

Во-первых потребуем теперь четыре типа операций: `l++`, `r++`, `l--`, `r--`.

Зная ответ для $[l_i, r_i]$, получить ответ для $[l_{i+1}, r_{i+1}]$ можно за $|l_{i+1} - l_i| + |r_{i+1} - r_i|$ операций.

Осталось перебирать запросы в правильном порядке, чтобы $\sum_i (|l_{i+1} - l_i| + |r_{i+1} - r_i|) \rightarrow \min$.

Чтобы получить правильный порядок, отсортируем отрезки по $\langle \lfloor \frac{l_i}{k} \rfloor, r_i \rangle$, где k – константа, которую ещё предстоит подобрать. После сортировки $|l_{i+1} - l_i| \leq 2k$, а r_i делятся на $\frac{n}{k}$ возрастающих групп, внутри каждой указатель r сделает $\leq n$ шагов. Итого $\Theta(mk + \frac{n}{k}n)$ операций.

Подбираем k : $f + g = \Theta(\max(f, g))$, при этом с ростом k $f = mk \nearrow$, $g = \frac{n}{k}n \searrow \Rightarrow$ оптимально взять k : $mk = \frac{n}{k}n \Rightarrow k = (n^2/m)^{1/2} = n/m^{1/2} \Rightarrow$ время работы $mk + \frac{n}{k}n = n\sqrt{m} + n\sqrt{m} = \Theta(n\sqrt{m})$.

Лекция #6: Сортировки

2 октября

6.1. Квадратичные сортировки

Def 6.1.1. Сортировка называется *стабильной*, если одинаковые элементы она оставляет в исходном порядке.

Пример: сортируем людей по имени. Люди с точки зрения сортировки считаются равными, если у них одинаковое имя. Тем не менее порядок людей в итоге важен. Во всех таблицах (гуглдок и т.д.) сортировки, которые вы применяете к данным, стабильные.

Def 6.1.2. Инверсия – пара $i < j: a_i > a_j$

Def 6.1.3. *Inv* – обозначение для числа инверсий в массиве

Lm 6.1.4. Массив отсортирован $\Leftrightarrow \text{Inv} = 0$

• Selection sort (сортировка выбором)

На каждом шаге выбираем минимальный элемент, ставим его в начале.

```
1 for (int i = 0; i < n; i++) {
2     j = index of min on [i..n);
3     swap(a[j], a[i]);
4 }
```

• Insertion sort (сортировка вставками)

Пусть префикс длины i уже отсортирован, возьмём a_i и вставим куда надо.

```
1 for (int i = 0; i < n; i++)
2     for (int j = i; j > 0 && a[j] > a[j - 1]; j--)
3         swap(a[j], a[j - 1]);
```

Корректность: по индукции по i ■

Заметим, что можно ускорить сортировку, место для вставки искать бинарным поиском.

Сортировка всё равно останется квадратичной.

• Bubble sort (сортировка пузырьком)

Бесполезна. Изучается, как дань истории. Простая.

```
1 for (int i = 0; i < n; i++)
2     for (int j = 1; j < n; j++)
3         if (a[j - 1] > a[j])
4             swap(a[j - 1], a[j]);
```

Корректность: на каждой итерации внешнего цикла очередной максимальный элемент встаёт на своё место, “всплывает”.

• Сравним пройденные сортировки.

Название	<	swap	stable
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	-
Insertion	$\mathcal{O}(n + \text{Inv})$	$\mathcal{O}(n + \text{Inv})$	+
Ins + B.S.	$\mathcal{O}(n \log n)$	$\mathcal{O}(n + \text{Inv})$	+
Bubble	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	+

Три нижние стабильны, т.к. `swap` применяется только к соседям, образующим инверсию. Количество `swap`-ов в `Insertion` равно `Inv`, т.к. каждый ровно на 1 уменьшает `Inv`.

Чем ценна сортировка выбором? `swap` может быть дорогой операцией. Пример: мы сортируем 10^3 тяжёлых для `swap` объектов, не имея дополнительной памяти.

Чем ценна сортировка вставками? Малая константа. Самая быстрая.

6.2. Оценка снизу на время сортировки

Пусть для сортировки объектов нам разрешено общаться с этими объектами единственным способом – сравнивать их на больше/меньше. Такие сортировки называются *основанные на сравнениях*.

Lm 6.2.1. Сортировка, основанная на сравнениях, делает на всех тестах $o(n \log n)$ сравнений $\Rightarrow \exists$ тест, на котором результат сортировки **не** корректен.

Доказательство. Докажем, что существует тест-перестановка. Всего есть $n!$ различных перестановок. Пусть сортировка делает не более k сравнений. Заставим её делать ровно k сравнений (возможно, несколько бесполезных). Результат каждого сравнения – меньше (0) или больше (1). Сортировка получает k бит информации, и результат её работы зависит только от этих k бит. То есть, если для двух перестановок она получит одни и те же k бит, одну из этих двух перестановок она отсортирует неправильно.

Сортировка корректна $\Rightarrow 2^k \geq n!$.

$2^k < n! \Rightarrow$ сортировка не корректна.

Осталось вспомнить, что $\log(n!) = \Theta(n \log n)$. ■

Мы доказали *нижнюю оценку* на время работы произвольной сортировки сравнениями. Доказали, что любая детерминированная (без использования случайных чисел) корректная сортировка делает хотя бы $\Omega(n \log n)$ сравнений.

6.3. Быстрые сортировки

Мы уже знаем одну сортировку за $\mathcal{O}(n \log n)$ – `HeapSort`.

Отметим её замечательные свойства: не использует дополнительной памяти, детерминирована.

6.3.1. CountSort (подсчётом)

Целые числа от 0 до $m - 1$ можно отсортировать за $\mathcal{O}(n + m)$.

В частности целые число от 0 до $2n$ можно отсортировать за $\mathcal{O}(n)$.

```

1 int n, a[n];
2 for (int i = 0; i < n; i++)  $\mathcal{O}(n)$ 
3     count[x]++; // насчитали, сколько раз x встречается в a
4 for (int x = 0; x < m; x++)  $\mathcal{O}(m)$ , перебрали x в порядке возрастания
5     while (count[x]--)
6         out(x);

```

Мы уже доказали, что сортировки, основанные на сравнениях не могут работать за $\mathcal{O}(n)$. В данном случае мы пользовались операцией `count[x]++`, ячейки массива `count` упорядочены также, как и числа. Именно это даёт ускорение.

6.4. Решение задачи по пройденным темам

Задача: даны два массива, содержащие множества, найти размер пересечения.

Решения:

1. Отсортировать первый массив, бинарным поиском найти элементы второго. $\mathcal{O}(n \log n)$.
2. Отсортировать оба массива, пройти двумя указателями. $\mathcal{O}(sort)$.
3. Элементы одного массива положить в хеш-таблицу, наличие элементов второго проверить. $\mathcal{O}(n)$, но требует $\Theta(n)$ допамяти, и имеет большую константу.

6.4.1. MergeSort (слиянием)

Идея: отсортируем левую половину массива, правую половину массива, сольём два отсортированных массива в один методом двух указателей.

```

1 void MergeSort( int l, int r, int *a, int *buffer ) { // [l, r)
2     if ( r - l <= 1 ) return;
3     int m = (l + r) / 2;
4     MergeSort(l, m, a, buffer);
5     MergeSort(m, r, a, buffer);
6     Merge(l, m, r, a, buffer); // слияние за  $\mathcal{O}(r-l)$ , используем буффер
7 }
```

`buffer` – дополнительная память, которая нужна функции `Merge`. Функция `Merge` берёт отсортированные куски $[l, m)$, $[m, r)$, запускает метод двух указателей, который отсортированное объединение записывает в `buffer`. Затем `buffer` копируется в `a[l, r)`.

Lm 6.4.1. Время работы $\mathcal{O}(n \log n)$

Доказательство. $T(n) = 2T(\frac{n}{2}) + n = \Theta(n \log n)$ ■

• Нерекурсивная версия без копирования памяти

Оставим ту же процедуру `Merge`, перепишем только рекурсивную функцию:

```

1 int n, *a, *buffer;
2 for (int k = 0; (1 << k) < n; k++)
3     for (int i = 0; i < n; i += 2 * (1 << k))
4         Merge(i, min(n, i + (1 << k)), min(n, i + 2 * (1 << k)), a, buffer)
5     swap(a, buffer);
6 return a; // результат содержится именно тут, указатель может отличаться от исходного a
```

6.4.2. QuickSort (быстрая)

Идея: выберем некий x , разобьём наш массив a на три части $< x$, $= x$, $> x$, сделаем два рекурсивных вызова, чтобы отсортировать первую и третью части. Утверждается, что сортировка будет быстро работать, если как x взять случайный элемент a

```

1 def QuickSort(a):
2     if len(a) <= 1: return a
3     x = random.choice(a)
4     b0 = select (< x).
5     b1 = select (= x).
6     b2 = select (> x).
7     return QuickSort(b0) + b1 + QuickSort(b2)
```

Этот псевдокод описывает общую идею, но обычно, чтобы `QuickSort` была реально быстрой сортировкой используют другую версию разделения массива на части.

Код 6.4.2. Быстрый `partition`.

```

1 void Partition( int l, int r, int x, int *a, int &i, int &j ) { // [l, r], x ∈ a[l, r]
2   i = l, j = r;
3   while (i <= j) {
4     while (a[i] < x) i++;
5     while (a[j] > x) j--;
6     if (i <= j) swap(a[i++], a[j--]);
7   }
8 }

```

Этот вариант разбивает отрезок $[l, r]$ массива a на части $[l, j](j, i)[i, r]$.

Замечание 6.4.3. $a[l, j] \leq x$, $a(j, i) = x$, $a[i, r] \geq x$

Замечание 6.4.4. Алгоритм не выйдет за пределы $[l, r]$

Доказательство. $x \in a[l, r]$, поэтому выполнится хотя бы один `swap`.

После `swap` верно $l < i \leq j < r$. Более того $a[l] \leq x$, $a[r] \geq x$. ■

Код 6.4.5. Собственно код быстрой сортировки:

```

1 void QuickSort( int l, int r, int *a ) { // [l, r]
2   if (l >= r) return;
3   int i, j;
4   Partition(l, r, a[random [l, r]], i, j);
5   QuickSort(l, j, a); // j < i
6   QuickSort(i, r, a); // j < i
7 }

```

6.4.3. Сравнение сортировок

Название	Время	space	stable
HeapSort	$\mathcal{O}(n \log n)$	$\Theta(1)$	-
MergeSort	$\Theta(n \log n)$	$\Theta(n)$	+
QuickSort	$\mathcal{O}(n \log n)$	$\Theta(\log n)$	-

Интересен вопрос существования стабильной сортировки, работающей за $\mathcal{O}(n \log n)$, не использующей дополнительную память. Среди уже изученных такой нет.

Такая сортировка существует. Она получается на основе `MergeSort` и `Merge` за $\mathcal{O}(n)$ без дополнительной памяти. На практике мы научимся делать `inplace stable Merge` за $\mathcal{O}(n \log n)$.

Лекция #7: Сортировки (продолжение)

9 октября

7.1. Quick Sort

• Глубина

Можно делать не два рекурсивных вызова, а только один, от меньшей части.

Тогда в худшем случае допмять = глубина = $\mathcal{O}(\log n)$.

Вместо второго вызова (l_2, r_2) сделаем $l = l_2$, $r = r_2$, `goto start`.

• Выбор x

На практике и в дз мы показали, что при любом детерминированном выборе x или даже как медианы элементов любых трёх фиксированных элементов, \exists тест, на котором время работы сортировки $\Theta(n^2)$. Чтобы на любом тесте QuickSort работал $\mathcal{O}(n \log n)$, нужно выбирать $x = a[\text{random } 1..r]$. Тем не менее, так как `random` – медленная функция, иногда для скорости пишут версию без `random`.

7.1.1. Оценка времени работы

Будем оценивать QuickSort, основанный на `partition`, который делит элементы на $(< x)$, x , $(> x)$. Также мы предполагаем, что все элементы различны.

• Доказательство #1.

Теорема 7.1.1. $T(n) \leq Cn \ln n$, где $C = 2 + \varepsilon$

Доказательство. Докажем по индукции.

$$T(n) = n + \frac{1}{n} \sum_{i=0..n-1} (T(i) + T(n-i-1))$$

Здесь среднее арифметическое берётся по всем вариантам выбора x среди всех элементов a . Чтобы оценить данную сумму, перейдём к интегралу.

$$T(n) = n + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \leq n + \frac{2}{n} \sum_{i=0}^{n-1} (Ci \ln i) \leq \frac{2}{n} C \int_1^n i \ln i di$$

Такой интеграл берётся по частям. Мы просто угадаем ответ $\frac{1}{2}x^2 \ln x - \frac{1}{4}x^2$.

$$T(n) \leq n + \frac{2C}{n} \left(\left(\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 \right) - \left(0 - \frac{1}{4} \right) \right) = n + Cn \ln n - \frac{2C}{4}n + \frac{2C}{4n} \stackrel{C \geq 2}{\leq} Cn \ln n \quad \blacksquare$$

• Доказательство #2.

Время работы вероятностного алгоритма — среднее арифметическое по всем рандомам. Время QuickSort пропорционально числу сравнений. Число сравнений – сумма по всем парам $i < j$ характеристической функции “сравнивали ли мы эту пару”, каждую пару мы сравним не более одного раза.

$$\frac{1}{R} \sum_{\text{random}} T(i) = \frac{1}{R} \sum_{\text{random}} \left(\sum_{i < j} is(i, j) \right) = \sum_{i < j} \left(\frac{1}{R} \sum_{\text{random}} is(i, j) \right) = \sum_{i < j} Pr[\text{сравнения}(i, j)]$$

Где Pr – вероятность. Осталось оценить вероятность. Для этого скажем, что у каждого элемента есть его индекс в отсортированном массиве.

Lm 7.1.2. $Pr[\text{сравнения}(i, j)] = \frac{2}{j-i+1}$ при $i < j$

Доказательство. Сравниться i и j могут только, если при некотором **Partition** выбор пал на один из них. Рассмотрим дерево рекурсии. Посмотрим на самую глубокую вершину, $[l, r)$ всё ещё содержит и i -й, и j -й элементы. Все элементы $i+1, i+2, \dots, j-1$ также содержатся (т.к. i и j – индексы в отсортированном массиве). i и j разделятся \Rightarrow **Partition** выберет один из $j-i+1$ элементов отрезка $[i, j]$. С вероятностью $\frac{2}{j-i+1}$ он совпадёт с i или j , тогда и только тогда i и j сравнятся. ■

Осталось посчитать сумму $\sum_{i < j} \frac{2}{j-i+1} = 2 \sum_i \sum_{j > i} \frac{1}{j-i+1} \leq 2(n \ln n + \Theta(n))$ ■

7.1.2. Introsort'97

На основе Quick Sort можно сделать быструю детерминированную сортировку.

1. Делаем Quick Sort от N элементов
2. Если $r - l$ не более 10, переключаемся на Insertion Sort
3. Если глубина более $3 \ln N$, переключаемся на Heap Sort

Такая сортировка называется Introsort, в C++:STL используется именно она.

7.2. Порядковые статистики

Задача поиска k -й порядковой статистики формулируется своим простейшим решением

```
1 int statistic(a, k) {
2     sort(a);
3     return a[k];
4 }
```

7.2.1. Одноветочный QuickSort

Вспомним реализацию Quick Sort 6.4.5

Quick Sort = выбрать x + Partition + 2 рекурсивных вызова Quick Sort.

Будем делать только 1 рекурсивный вызов:

Код 7.2.1. Порядковая статистика

```
1 int Statistic( int l, int r, int *a, int k ) { // [l, r]
2     if (r <= l) return;
3     int i, j;
4     Partition(l, r, a[random [l, r]], i, j);
5     if (j < k && k < i) return x;
6     return k <= j ? Statistic(l, j, a) : Statistic(i, r, a);
7 }
```

Действительно, зачем вызываться от второй половины, если ответ находится именно в первой?

Теорема 7.2.2. Время работы 7.2.1 равно $\Theta(n)$

Доказательство. С вероятностью $\frac{1}{3}$ мы попадем в элемент, который лежит во второй трети отсортированного массива. Тогда после Partition размеры кусков будут не более $\frac{2}{3}n$. Если же не попали, то размеры не более n , вероятность этого $\frac{2}{3}$. Итого:

$$T(n) = n + \frac{1}{3}T\left(\frac{2}{3}n\right) + \frac{2}{3}T(n) \Rightarrow T(n) = 3n + T\left(\frac{2}{3}n\right) \leq 9n = \Theta(n) \quad \blacksquare$$

Замечание 7.2.3. Мы могли бы повторить доказательство 7.1.1, тогда нам нужно было бы оценить сумму $\sum T(\max(i, n - i - 1))$. Это технически сложнее, зато дало бы константу 4.

7.2.2. Детерминированный алгоритм

Statistic = выбрать x + Partition + 1 рекурсивный вызов Statistic.

Чтобы этот алгоритм стал детерминированным, нужно хорошо выбирать x .

- **Идея.** Разобьем n элементов на группы по 5 элементов, в каждой группе выберем медиану, из полученных $\frac{n}{5}$ медиан выберем медиану, это и есть x .

Утверждение 7.2.4. На массиве длины 5 медиану можно выбрать за 6 сравнений.

Поскольку из $\frac{n}{5}$ меньшие $\frac{n}{10}$ не больше x , хотя бы $\frac{3}{10}n$ элементов исходного массива **не более** выбранного x . Аналогично хотя бы $\frac{3}{10}n$ элементов **не менее** выбранного x . Это значит, что после Partition размеры кусков не будут превосходить $\frac{7}{10}n$. Теперь оценим время работы алгоритма:

$$T(n) \leq 6\frac{n}{5} + T\left(\frac{n}{5}\right) + n + T\left(\frac{7}{10}n\right) = 2.2\left(n + \frac{9}{10}n + \left(\frac{9}{10}\right)^2n + \dots\right) = 22n = \Theta(n) \quad \blacksquare$$

7.2.3. C++

В C++: STL есть следующие функции

1. `nth_element(a, a + k, a + n)` – k -я статистика на основе одноветочного Quick Sort. После вызова функции k -я статистика стоит на своём месте, слева меньшие, справа большие.
2. `partition(a, a + n, predicate)` – Partition по произвольному предикату.

7.3. Integer sorting

За счёт чего получается целые числа сортировать быстрее чем произвольные объекты?

$\forall k$ операция деления нацело на k : $x \rightarrow \lfloor \frac{x}{k} \rfloor$ сохраняет порядок.

Если мы хотим сортировать вещественные числа, данные с точностью $\pm \varepsilon$, их можно привести к целым: домножить на $\frac{1}{\varepsilon}$ и округлить, после чего сортировать целые.

7.3.1. Count sort

Давайте используем уже известный нам Count Sort, чтобы стабильно отсортировать пары $\langle a_i, b_i \rangle$

```

1 void CountSort(int n, int *a, int *b) { // 0 <= a[i] < m
2     for (int i = 0; i < n; i++)
3         count[a[i]]++; // сколько раз встречается
4     // pos[i] -- позиция начала куска ответа, состоящего из пар <i, ?>
5     for (int i = 0; i + 1 < m; i++)
6         pos[i + 1] = pos[i] + count[i];
7     for (int i = 0; i < n; i++)
8         result[pos[a[i]]++] = {a[i], b[i]}; // нужна допамать!
9 }
```

Важно то, что сортировка **стабильна**, из этого следует наш следующий алгоритм:

7.3.2. Radix sort

Отсортируем n строк длины L . Символ строки – целое число из $[0, k)$.

- **Алгоритм:** отсортируем сперва по последнему символу, затем по предпоследнему и т.д.
- **Корректность:** мы сортируем стабильной сортировкой строки по символу номер i , строки уже отсортированы по символам $(i, L]$. Из стабильности имеем, что строки равные по i -му символу будут отсортированы как раз по $(i, L] \Rightarrow$ теперь строки отсортированы по $[i, L]$.
- **Время работы:** мы L раз вызвали сортировку подсчётом $\Rightarrow \mathcal{O}(L(n + k))$.

Теперь заметим, что $\forall k$ число из $[0, m)$ – строка длины $\log_k m$ над алфавитом $[0, k)$.

При $k = n$ получаем время работы $n \lceil \log_n m \rceil$.

7.3.3. Bucket sort

Главная идея заключается в том, чтобы числа от `min` до `max` разбить на n бакетов (пакетов, карманов, корзины). Числовая прямая бьётся на n отрезков равной длины, i -й отрезок:

$$\left[\min + \frac{i}{n}(\max - \min + 1), \min + \frac{i+1}{n}(\max - \min + 1) \right)$$

Каждое число x_j попадает в отрезок номер $i_j = \lfloor \frac{x_j - \min}{\max - \min + 1} n \rfloor$. Бакеты уже упорядочены: все числа в 0-м меньше всех чисел в 1-м и т.д. Осталось упорядочить числа внутри бакетов. Это

можно сделать или вызовом Insertion Sort (алгоритм В.І.), чтобы минимизировать константу, или рекурсивным вызовом Bucket Sort (алгоритм В.В.)

Код 7.3.1. Bucket Sort

```

1 void BB( vector<int> &a ) {
2     if (a.empty()) return;
3     int min = *min_element(a.begin(), a.end());
4     int max = *max_element(a.begin(), a.end());
5     if (min == max) return; // уже отсортирован
6     vector<int> b[n];
7     for (int x : a) {
8         int i = (long long)n * (x - min) / (max + min + 1); // номер бакета
9         b[i].push_back(x);
10    }
11    a.clear();
12    for (int i = 0; i < n; i++) {
13        BB(b[i]); // отсортировали каждый бакет рекурсивным вызовом
14        for (int x : b[i]) a.push_back(x); // сложили результат в массив a
15    }
16 }
```

Lm 7.3.2. $\max - \min \leq n \Rightarrow$ и BB, и VI работают за $\Theta(n)$

Доказательство. В каждом рекурсивном вызове $\max - \min \leq 1$ ■

Lm 7.3.3. BB работает за $\mathcal{O}(n \lceil \log(\max - \min) \rceil)$

Доказательство. Ветвление происходит при $n \geq 2 \Rightarrow$ длина диапазона сокращается как минимум в два раза \Rightarrow глубина рекурсии не более \log . На каждом уровне рекурсии суммарно не более n элементов. ■

Замечание 7.3.4. На самом деле ещё быстрее, так как уменьшение не в 2 раза, а в n раз.

Lm 7.3.5. На массиве, сгенерированном равномерным распределением, время VI = $\Theta(n)$

Доказательство. Время работы VI: $T(n) = \frac{1}{R} \sum_{random} (\sum_i k_i^2)$, где k_i – размер i -го бакета.

Заметим, что $\sum_i k_i^2$ – число пар элементов, у которых совпал номер бакета.

$$\frac{1}{R} \sum_{random} (\sum_i k_i^2) = \frac{1}{R} \sum_{random} (\sum_{j_1=1}^n \sum_{j_2=1}^n [i_{j_1} == i_{j_2}]) = \sum_{j_1=1}^n \sum_{j_2=1}^n (\frac{1}{R} \sum_{random} [i_{j_1} == i_{j_2}]) = \sum_{j_1=1}^n \sum_{j_2=1}^n \Pr[i_{j_1} == i_{j_2}]$$

Осталось посчитать вероятность, при $j_1 = j_2$ получаем 1, при $j_1 \neq j_2$ получаем $\frac{1}{n}$ из равномерности распределения $T(n) = n \cdot 1 + n(n-1) \cdot \frac{1}{n} = 2n - 1 = \Theta(n)$. Получили точное среднее время работы VI на случайных данных. ■

7.4. Kirkpatrick'84 sort

Научимся сортировать n целых чисел из промежутка $[0, C)$ за $\mathcal{O}(n \log \log C)$.

Пусть $2^{2^{k-1}} < C \leq 2^{2^k}$, округлим вверх до 2^{2^k} ($\log \log C$ увеличился не более чем на 1).

Если числа достаточно короткие, отсортируем их подсчётом, иначе каждое 2^k -битное число x_i представим в виде двух 2^{k-1} -битных половин: $x_i = \langle a_i, b_i \rangle$.

Отсортируем отдельно a_i и b_i рекурсивными вызовами.

```

1 vector<int> Sort(int k, vector<int> &x) {
2     int n = x.size()
3     if (n >= 2^{2^k}) return CountSort(x) // за O(n)
4     vector<int> a(n), b(n), as, result;
5     unordered_map<int, vector<int>> A; // хеш-таблица
6     for (int i = 0; i < n; i++) {
7         a[i] = старшие 2^{k-1} бит x[i];
8         b[i] = младшие 2^{k-1} бит x[i];
9         A[a[i]].push_back(b[i]); // для каждого a[i] храним все парные с ним b[i]
10    }
11    // построим список ключей хеш-таблицы, список всех a[i]
12    for (auto &p : A) as.push_back(p.first);
13    as = Sort(k - 1, as); // отсортировали все a[i]
14    for (int a : as) {
15        vector<int> &bs = A[a]; // теперь нужно отсортировать вектор bs
16        int i = max_element(bs.begin(), bs.end()) - bs.begin(), max_b = bs[i];
17        swap(bs[i], bs.back()), bs.pop_back(); // удалили максимальный элемент
18        bs = Sort(k - 1, bs); // отсортировали всё кроме максимума
19        for (int b : bs) result.push_back(<a, b>); // выписали результат без максимума
20        result.push_back(<a, max_b>); // отдельно добавили максимальный элемент
21    }
22    return result;
23 }
```

Оценим время работы. $T(k, n) = n + \sum_i T(k - 1, m_i)$. m_i – размеры подзадач, рекурсивных вызовов. Вспомним, что мы из каждого списка bs выкинули 1 элемент, максимум. Поэтому:

$$\sum m_i = |as| + \sum_a (|bs_a| - 1) = \sum_a |bs_a| = n$$

Глубина рекурсии не более k , на каждом уровне рекурсии суммарный размер всех подзадач не более n . Поэтому суммарное время работы $\mathcal{O}(nk) = \mathcal{O}(n \log \log C)$.

Жаль, но на практике из-за большой константы хеш-таблицы преимущества мы не получим.

Лекция #8: Кучи

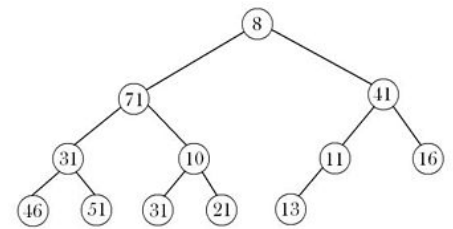
16 октября

8.1. Min-Max Heap (Atkison'86)

Min-Max куча – Inplace структура данных, которая строится на исходном массиве и умеет делать **Min**, **Max** за $\mathcal{O}(1)$, а также **Add** и **ExtractMin** за $\mathcal{O}(\log n)$.

Заметим, что мы могли бы просто завести две кучи, одну на минимум, вторую на максимум, а между ними хранить обратные ссылки. Минусы этого решения – в два раза больше памяти, примерно в два раза больше константа времени.

Дети в Min-Max куче такие же, как в бинарной $i \rightarrow 2i, 2i + 1$. Инвариант: на каждом нечётном уровне хранится минимум в поддереве, на каждом чётном максимум в поддереве. В корне $h[1]$ хранится минимум. Максимум считается как $\max(h[2], h[3])$. Операции **Add** и **ExtractMin** также, как в бинарной куче выражаются через **SiftUp**, **SiftDown**.



• SiftUp

Предположим, что вершина v , которую нам нужно поднять, находится на уровне минимумов (противоположный случай аналогичен). Тогда $\frac{v}{2}$, отец v , находится на уровне максимумов. Возможны следующие ситуации:

1. Значение у v не больше, чем у отца, тогда делаем обычный SiftUp с шагом $i \rightarrow \frac{i}{4}$.
2. Иначе меняем местами v и $\frac{v}{2}$, и из $\frac{v}{2}$ делаем обычный SiftUp с шагом $i \rightarrow \frac{i}{4}$.

Время работы, очевидно, $\mathcal{O}(\log n)$. Алгоритм сделает не более чем $\frac{n}{2} + 1$ сравнение, то есть, примерно в два раза быстрее SiftUp от обычной бинарной кучи.

• Корректность SiftUp.

Если нет конфликта с отцом, то вся цепочка от i с шагом два $(i, \frac{i}{4}, \frac{i}{16}, \dots)$ не имеет конфликтов с цепочкой с шагом два от отца. После **swap** внутри SiftUp конфликт не появится. Если же с отцом был конфликт, то после **swap**($v, \frac{v}{2}$) у $\frac{v}{2}$ и его отца, $\frac{v}{4}$, конфликта нет. ■

• SiftDown

Предположим, что вершина v , которую нам нужно спустить, находится на уровне минимумов (противоположный случай аналогичен). Тогда дети v находятся на уровне максимумов. Возможны следующие ситуации:

1. У v меньше 4 внуков. Обработает случай руками за $\mathcal{O}(1)$.
2. Среди внуков v есть те, что меньше v . Тогда найдём наименьшего внука v и поменяем его местами с v . Осталось проверить, если на новом месте v конфликтует со своим отцом, поменять их местами. Продолжаем SiftDown из места, где первоначально был наименьший внук v .
3. У v все внуки неменьше. Тогда ничего исправлять не нужно.

На каждой итерации выполняется 5 сравнений – за 4 выберем минимум из 5 элементов, ещё за 1 решим возможный конфликт с отцом. После этого глубина уменьшается на 2. Итого $\frac{5}{2} \log_2 n + \mathcal{O}(1)$ сравнений. Что чуть больше, чем у обычной бинарной кучи ($2 \log_2 n$ сравнений).

MinMax кучу можно построить за линейное время inplace аналогично двоичной куче.

8.2. Leftist Heap (Clark'72)

Пусть в корневом дереве каждая вершина имеет степень 0, 1 или 2. Введём для каждой вершины

Def 8.2.1. $d(v)$ – расстояние вниз от v до ближайшего отсутствия вершины.

Заметим, что $d(\text{NULL}) = 0$, $d(v) = \min(d(v.\text{left}), d(v.\text{right})) + 1$

Lm 8.2.2. $d(v) \leq \log_2(\text{size} + 1)$

Доказательство. Заметим, что полное бинарное дерево высоты $d(v)-1$ является нашим поддеревом $\Rightarrow \text{size} \geq 2^{d(v)} - 1 \Leftrightarrow \text{size} + 1 \geq 2^{d(v)}$ ■

Def 8.2.3. Левацкая куча (leftist heap) – структура данных в виде бинарного дерева, в котором в каждой вершине один элемент. Для этого дерева выполняются условие кучи и условие leftist: $\forall v \quad d(v.\text{left}) \geq d(v.\text{right})$

Следствие 8.2.4. В левацкой куче $\log_2 n \geq d(v) = d(v.\text{right}) + 1$

Главное преимущество левацких куч над предыдущими – возможностью быстрого слияния (Merge). Через Merge выражаются Add и ExtractMin (слияние осиротевших детей).

• Merge

Для удобства реализации EMPTY – пустое дерево, которое имеет $\text{left} = \text{EMPTY}$, $\text{right} = \text{EMPTY}$, $x = +\infty$, $d = 0$.

```

1 Node* Merge(Node* a, Node* b) {
2   if (a->x >= b->x) swap(a, b); // теперь a - общий корень
3   if (b == EMPTY) return a; // теперь обе кучи непусты
4   a->right = Merge(a->right, b);
5   if (a->right->d > a->left->d) // если нарушен инвариант leftist
6     swap(a->right, a->left); // исправили инвариант leftist
7   a->d = a->right->d + 1; // обновили d
8   return a;
9 }
```

Время работы: на каждом шаге рекурсии величина $a \rightarrow d + b \rightarrow d$ уменьшается $\Rightarrow \text{time} \leq a \rightarrow d + b \rightarrow d \leq 2 \log_2 n$.

8.3. Skew Heap (Tarjan'86)

Уберём условие $d(v.\text{left}) \geq d(v.\text{right})$. В функции Merge уберём 5 и 7 строки. То есть, в Merge мы теперь не храним d , а просто всегда делаем swap детей. Полученная куча называется “скошенной” (skew heap). В худшем случае один Merge теперь может работать $\Theta(n)$, но мы докажем амортизированную сложность $\mathcal{O}(\log_2 n)$. Скошенная куча выгодно отличается длиной реализации и константой времени работы.

• Доказательство времени работы

Def 8.3.1. Пусть v – вершина, p – её отец, size – размер поддерева. Ребро $p \rightarrow v$ называется Тяжёлым если $\text{size}(v) > \frac{1}{2}\text{size}(p)$
Лёгким если $\text{size}(v) \leq \frac{1}{2}\text{size}(p)$

Def 8.3.2. Ребро $p \rightarrow v$ называется правым, если v – правый сын p .

Заметим, что из вершины может быть не более 1 тяжёлого ребра вниз.

Lm 8.3.3. На любом вертикальном пути не более $\log_2 n$ лёгких рёбер.

Доказательство. При спуске по лёгкому ребру размер поддерева меняется хотя бы в 2 раза. ■

Как теперь оценить время работы **Merge**? Нужно чем-то амортизировать количество тяжёлых рёбер. Введём потенциал φ = “количество правых тяжёлых рёбер”.

Теорема 8.3.4. Время работы **Merge** в среднем $\mathcal{O}(\log n)$

Доказательство. Разделим время работы i -й операции на две части – количество лёгких и тяжёлых рёбер.

$$t_i = \mathbb{L}_i + \mathbb{T}_i \leq \log_2 n + \mathbb{T}_i$$

Теперь распишем изменения потенциала φ . Для этого заметим, что если мы прошли вправо по лёгкому ребру, то количество правых тяжёлых увеличилось после **swap** на 0 или на 1, если же мы прошли вправо по тяжёлому ребру, то количество правых тяжёлых после **swap** точно уменьшилось на 1.

$$\Delta\varphi \geq \log_2 n - \mathbb{T}_i \Rightarrow a_i = t_i + \Delta\varphi \leq 2\log n$$

Осталось заметить, что $0 \leq \varphi \leq n - 1$, поэтому среднее время работы $\mathcal{O}(\log n)$. ■

8.4. Спско-куча

Будем хранить элементы кучи в двусвязном списке и поддерживать указатель на текущий минимум. Заметим, что операции **Merge**, **Add**, **DecreaseKey**, **GetMin** в списке работают за $\mathcal{O}(1)$.

Операция **ExtractMin** состоит из **GetMin** за $\mathcal{O}(1)$, удаления из списка за $\mathcal{O}(1)$ и самой сложной части “найти новый минимум”, которую мы пока умеем делать лишь за $\mathcal{O}(n)$.

Хочется линейный проход по всем элементам амортизировать уменьшением длины списка. Например, можно все элементы разбить по группы по \sqrt{n} элементов, подробнее можно посмотреть в [разборе домашнего задания](#).

8.5. Van Embde Boas’75 trees

Куча над целыми числами из $[0, C)$, умеющая всё, что подобает уметь куче, за $\mathcal{O}(\log \log C)$.

При описании кучи есть четыре принципиально разных случая:

1. Мы храним пустое множество
2. Мы храним ровно одно число
3. $C \leq 2$
4. Мы храним хотя бы два числа, $C > 2$.

Первые три вы разберёте самостоятельно, здесь же детально описан **только 4-й случай**.

Пусть $2^{2^{k-1}} < C \leq 2^{2^k}$, округлим C вверх до 2^{2^k} ($\log \log C$ увеличился не более чем на 1).

Основная идея – промежуток $[0, C)$ разбить на \sqrt{C} кусков длины \sqrt{C} . Также, как и в **BucketSort**, i -й кусок содержит числа из $[i\sqrt{C}, (i+1)\sqrt{C})$. Заметим, $\sqrt{C} = \sqrt{2^{2^k}} = 2^{2^{k-1}}$.

Теперь опишем кучу уровня $k \geq 1$, **Heap<k>**, хранящую числа из $[0, 2^{2^k})$.

```

1 struct Heap<k> {
2     int min, size; // отдельно храним минимальный элемент и размер
3     Heap<k-1>* notEmpty; // номера непустых кусков
4     unordered_map<int, Heap<k-1>*> parts; // собственно куски
5 };

```

• Как добавить новый элемент?

Номер куска по числу x : $\text{index}(x) = (x \gg 2^{\{k-1\}})$;

```

1 void Heap<k>::add(int x) {
2     // size ≥ 2, k ≥ 2, разбираем только интересный случай
3     int i = index(x);
4     if parts[i] is empty
5         notEmpty->add(i); // появился новый непустой кусок, делаем рекурсивный вызов
6         parts[i]->add(x); // O(1)
7     else
8         parts[i]->add(x); // рекурсивный вызов
9     size++.
10    min = parts[notEmpty->min()]->min(); // универсальный способ пересчитать минимум, O(1)
11 }

```

Время работы равна глубине рекурсии $= \mathcal{O}(k) = \mathcal{O}(\log \log C)$.

• Как удалить элемент?

```

1 void Heap<k>::del(int x) {
2     // size ≥ 2, k ≥ 2, разбираем только интересный случай
3     int i = index(x);
4     if parts[i]->size == 1
5         notEmpty->del(i); // кусок стал пустым, делаем рекурсивный вызов
6         parts[i] = empty heap
7     else
8         parts[i]->del(i)
9     min = parts[notEmpty->min()]->min(); // универсальный способ пересчитать минимум, O(1)
10 }

```

Время работы и анализ такие же, как при добавлении. Получается, мы можем удалить не только минимум, а произвольный элемент по значению за $\mathcal{O}(\log \log C)$.

Жаль, но на практике из-за большой константы хеш-таблицы преимущества мы не получим.

Лекция #9: Кучи

23 октября

9.1. Нижняя оценка на построение бинарной кучи

Мы уже умеем давать нижние оценки на число сравнений во многих алгоритмах. Везде это делалось по одной и той же схеме, например, для сортировки “нам нужно различать $n!$ перестановок, поэтому нужно сделать хотя бы $\log(n!) = \Theta(n \log n)$ сравнений”.

В случае построения бинарной кучи от перестановки, ситуация сложнее. Есть несколько возможных корректных ответов. Собственно любая корректная куча может быть ответом. Обозначим за $H(n)$ количество перестановок, являющихся корректной бинарной кучей.

Лм 9.1.1. $H(n) = \frac{n!}{\prod_i size_i}$, где $size_i$ – размер поддерева i -й вершины кучи

Доказательство. $H(n) = \binom{n-1}{l} H(l) H(r) = \frac{n!}{l! r! n} H(l) H(r)$, где l, r – размеры детей, $l + r + 1 = n$.

Аналогично $H(l) = \frac{l!}{ll! lr! l} H(ll) H(lr)$, где ll, lr – размеры левых внуков.

Аналогично $H(r) = \frac{r!}{rl! rr! r} H(rl) H(rr)$, где rl, rr – размеры правых внуков.

Подставляем, получаем $H(n) = \frac{n!}{l! r! n} \frac{l!}{ll! lr! l} \frac{r!}{rl! rr! r} H(rl) H(rr) H(ll) H(lr) = \frac{n!}{n \cdot l \cdot r} \frac{H(ll)}{ll!} \frac{H(lr)}{lr!} \frac{H(rl)}{rl!} \frac{H(rr)}{rr!}$

Подставляя внуков и т.д. получим формулу, которую доказываем. ■

Теорема 9.1.2. Любой корректный алгоритм построения бинарной кучи делает в худшем случае не менее $1.364n$ сравнений

Доказательство. Пусть алгоритм делает k сравнений, тогда он разбивает $n!$ перестановок на 2^k классов. Класс – те перестановки, на которых наш алгоритм одинаково сработает. Заметим, что алгоритм делающий одно и то же с разными перестановками, на выходе даст разные перестановки. Если x_i – количество элементов в i -м классе, корректный алгоритм переведёт эти x_i перестановок в x_i различных бинарных куч. Поэтому

$$x_i \leq H(n)$$

Из $\sum_{i=1..2^k} x_i = n!$ имеем $\max_{i=1..2^k} x_i \geq \frac{n!}{2^k}$. Итого:

$$H(n) \geq \max x_i \geq \frac{n!}{2^k} \Rightarrow \frac{n!}{\prod_i size_i} \geq \frac{n!}{2^k} \Rightarrow 2^k \geq \prod size_i \Rightarrow k \geq \sum \log size_i$$

Рассмотрим случай полного бинарного дерева $n = 2^k - 1 \Rightarrow$

$$\sum \log size_i = (\log 3) \frac{n+1}{4} + (\log 7) \frac{n+1}{8} + (\log 15) \frac{n+1}{16} + \dots = (n+1) \left(\frac{\log 3}{4} + \frac{\log 7}{8} + \frac{\log 15}{16} + \dots \right).$$

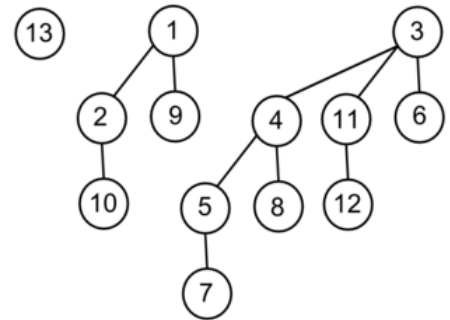
При $n \rightarrow +\infty$ величина $\frac{\sum \log size_i}{n+1}$ имеет предел, вычисление первых 20 слагаемых даёт $1.36442\dots$ и ошибку в 5-м знаке после запятой. ■

9.2. Биномиальная куча (Vuillemin'78)

9.2.1. Основные понятия

Определим понятие “биномиальное дерево” рекурсивно.

Def 9.2.1. Биномиальным деревом ранга 0 или T_0 будем называть одну вершину. Биномиальным деревом ранга $n+1$ или T_{n+1} будем называть дерево T_n , к корню которого подвесили еще одно дерево T_n (порядок следования детей не важен). При этом биномиальное дерево должно удовлетворять свойству кучи (значение в вершине не меньше значения в предках).



Выпишем несколько простых свойств биномиальных деревьев.

Lm 9.2.2. $|T_n| = 2^n$

Доказательство. Индукция по рангу дерева (далее эта фраза и проверка базы индукции будет опускаться). База: для $n = 0$ дерево состоит из одной вершины.

Переход: $|T_{n+1}| = |T_n| + |T_n| = 2^n + 2^n = 2^{n+1}$. ■

Lm 9.2.3. $\text{degRoot}(T_n) = n$

Доказательство. $\text{degRoot}(T_{n+1}) = \text{degRoot}(T_n) + 1 = n + 1$ ■

Lm 9.2.4. Сыновьями T_n являются деревья T_0, T_1, \dots, T_{n-1} .

Доказательство. Сыновья T_{n+1} – все сыновья T_n , т.е. T_0, \dots, T_{n-1} , и новый T_n . ■

Lm 9.2.5. $\text{depth}(T_n) = n$

Доказательство. $\text{depth}(T_{n+1}) = \max(\text{depth}(T_n), 1 + \text{depth}(T_n)) = 1 + \text{depth}(T_n) = 1 + n$ ■

• Как хранить биномиальное дерево?

```
1 struct Node{
2     Node *next, *child, *parent;
3     int x, rank;
4 };
```

Здесь **child** – ссылка на первого сына, **next** – ссылка на брата, **x** – полезные данные, которые мы храним. Список сыновей вершины **v**: **v->child**, **v->child->next**, **v->child->next->next**, ...

Теперь определим понятие “биномиальная куча”.

Def 9.2.6. Биномиальная куча – список биномиальных деревьев различного ранга.

У любого числа n есть единственное представление в двоичной системе счисления $n = \sum_i 2^{k_i}$. В биномиальной куче из n элементов деревья будут иметь размеры как раз 2^{k_i} . Заметим, что в списке не более $\log_2 n$ элементов.

9.2.2. Операции с биномиальной кучей

Add и ExtractMin выражаются, также как и в левацкой куче, через Merge. Чтобы выразить ExtractMin заметим, что дети корня любого биномиального дерева по определению являются биномиальной кучей. То есть, после удаления минимума нужно сделать Merge от кучи, образованной детьми удалённой вершины и оставшихся деревьев.

DecreaseKey – обычный SiftUp, работает по лемме 9.2.5 за $\mathcal{O}(\log n)$.

В чём проблема Merge, почему просто не соединить два списка? После соединения появятся биномиальные деревья одинакового ранга. К счастью, по определению мы можем превратить их в одно дерево большего ранга

```

1 Node* join(Node* a, Node* b) { // a->rank == b->rank
2   if (a->x > b->x) swap(a, b);
3   b->next = a->child, a->child = b; // добавили b в список детей a
4   return a;
5 }
```

Теперь пусть у нас есть список с деревьями возможно одинакового ранга. Отсортируем их по рангу и будем вызывать join, пока есть деревья одного ранга.

```

1 list<Node*> Normalize( list<Node*> &a ) {
2   list<Node*> roots[maxRank + 1], result;
3   for (Node* v : a) roots[v->rank].push_back(v);
4   for (int rank = 0; rank <= maxRank; rank++) {
5     while (roots[rank].size() >= 2) {
6       Node* a = roots[rank].back(); roots[rank].pop_back();
7       Node* b = roots[rank].back(); roots[rank].pop_back();
8       roots[rank + 1].push_back(join(a, b));
9     }
10  }
11  return result;
12 }
```

Время работы Normalize равно $|a| + \max Rank = \mathcal{O}(\log n)$.

9.2.3. Add и Merge за $\mathcal{O}(1)$

У нас уже полностью описана классическая биномиальная куча. Давайте её ускорять. Уберём условие на “все ранги должны быть различны”. То есть, новая куча – список произвольных биномиальных деревьев. Теперь Add, Merge, GetMin, очевидно, работают за $\mathcal{O}(1)$. Но ExtractMin теперь работает за длину списка. Вызовем после ExtractMin процедуру Normalize, которая укоротит список до $\mathcal{O}(\log_2 n)$ корней. Теперь время ExtractMin самортизируется потенциалом $\varphi = \text{Roots}$ (длина списка, количество корней).

Теорема 9.2.7. Среднее время работы ExtractMin равно $\mathcal{O}(\log n)$

Доказательство. $t_i = \text{Roots} + \max Rank$, $\Delta\varphi \leq \log_2 n - \text{Roots} \Rightarrow a_i = t_i + \Delta\varphi = \mathcal{O}(\log n)$.

Заметим также, $0 \leq \varphi \leq n \Rightarrow$ среднее время ExtractMin $\mathcal{O}(\log n)$. ■

9.3. Куча Фибоначчи (Fredman, Tarjan’84)

Отличается от всех вышеописанных куч тем, что умеет делать DecreaseKey за $\mathcal{O}(1)$. Является

апгрейдом биномиальной кучи. Собственно биномиальные кучи практической ценности не имеют, они во всём хуже левацкой кучи, а нужны они нам, как составная часть кучи Фибоначчи.

Если `DecreaseKey` будет основан на `SiftUp`, как ни крути, быстрее $\log n$ он работать не будет. Нужна новая идея для `DecreaseKey`, вот она: отрежем вершину со всем её поддеревом и поместим в список корней. Чтобы “отрезать” за $\mathcal{O}(1)$ кроме ссылки на отца теперь ещё нужно хранить двусвязный список детей (`left`, `right`).

```
1 struct Node{
2     Node *child, *parent, *left, *right;
3     int x, degree; // ранг биномиального дерева равен степени
4     bool marked; // удаляли ли мы уже сына у этой вершины
5 };
```

• Пометки `marked`

Чтобы деревья большого ранга оставались большого размера, нужно запретить удалять у них много детей. Скажем, что `marked` – флаг, равный единице, если мы уже отрезали сына вершины. Когда мы пытаемся отрезать у v второго сына, отрежем рекурсивно и вершину v тоже.

```
1 list<Node*> heap; // куча - список корней биномиальных деревьев
2 void CascadingCut(Node *v) {
3     Node *p = v->parent;
4     if (p == NULL) return; // мы уже корень
5     p->degree--; // поддерживаем степень, будем её потом использовать, как ранг
6     <Delete v from linked list of children of p>
7     heap.push_back(v), v->marked = 0; // начнём новую жизнь в качестве корня!
8     if (p->parent && p->marked++) // если папа - корень, ничего не нужно делать
9         CascadingCut(p); // у p только что отрезали второго сына, пора сделать его корнем
10 }
11 void DecreaseKey(int i, int x) { // i - номер элемента
12     pos[i]->x = x; // обратная ссылка
13     CascadingCut(i);
14 }
```

Важно заметить, что когда вершина v становится корнем, её `mark` обнуляется, она обретает новую жизнь, как корневая вершина ранга $v->degree$.

Def 9.3.1. Ранг вершины в Фибоначчиевой куче – её степень на тот момент, когда вершина последний раз была корнем.

Если мы ни разу не делали `DecreaseKey`, то `rank = degree`. В общем случае:

Lm 9.3.2. $v.\text{rank} = v.\text{degree} + v.\text{mark}$

Заметим, что ранги нам нужны только в `Normalize`, то есть, в тот момент, когда вершина является корнем.

Теорема 9.3.3. `DecreaseKey` работает в среднем за $\mathcal{O}(1)$

Доказательство. `Marked` – число помеченных вершин. Пусть $\varphi = \text{Roots} + 2\text{Marked}$.

Амортизированное время операций кроме `DecreaseKey` не менялось, так как они не меняют `Marked`. Пусть `DecreaseKey` отрезал $k + 1$ вершину, тогда $\Delta\text{Marked} \leq -k$, $\Delta\text{Roots} \leq k + 1$, $a_i = t_i + \Delta\varphi = t_i + \Delta\text{Roots} + 2\Delta\text{Marked} \leq (k + 1) + (k + 1) - 2k = \Theta(1)$. ■

9.3.1. Фибоначчиевы деревья

Чтобы оценка $\mathcal{O}(\log n)$ на `ExtractMin` не испортилась нам нужно показать, что $size(rank)$ – всё ещё экспоненциальная функция.

Def 9.3.4. *Фибоначчиево дерево F_n – биномиальное дерево T_n , с которым произвели рекурсивное обрезание: отрезали не более одного сына, и рекурсивно запустились от выживших детей.*

Оценим S_n – минимальный размер дерева F_n

Lm 9.3.5. $\forall n \geq 2: S_n = 1 + S_0 + S_1 + \dots + S_{n-2}$

Доказательство. Мы хотим минимизировать размер.

Отрезать ли сына? Конечно, да! Какого отрезать? Самого толстого, то есть, F_{n-1} . ■

Заметим, что полученное рекуррентное соотношение верно также и для чисел Фибоначчи:

Lm 9.3.6. $\forall n \geq 2: Fib_n = 1 + Fib_0 + Fib_1 + \dots + Fib_{n-2}$

Доказательство. Индукция. Пусть $Fib_{n-1} = Fib_0 + Fib_1 + \dots + Fib_{n-3}$, тогда

$$1 + Fib_0 + Fib_1 + \dots + Fib_{n-2} = (1 + Fib_0 + Fib_1 + \dots + Fib_{n-3}) + Fib_{n-2} = Fib_{n-1} + Fib_{n-2} = Fib_n \quad \blacksquare$$

Lm 9.3.7. $S_n = Fib_n$

Доказательство. База: $Fib_0 = S_0 = 1, Fib_1 = S_1 = 1$. Формулу перехода уже доказали. ■

Получили оценку снизу на размер дерева Фибоначчи ранга n : $S_n \geq \frac{1}{\sqrt{5}}\varphi^n$, где $\varphi = \frac{1+\sqrt{5}}{2}$.

И поняли, почему куча называется именно Фибоначчиевой.

9.3.2. Завершение доказательства

Фибоначчиева куча – список деревьев. Эти деревья **не являются Фибоначчиевыми** по нашему определению. Но размер дерева ранга k не меньше S_k .

Покажем, что новые деревья, которые мы получаем по ходу операций `Normalize` и `DecreaseKey` не меньше Фибоначчиевых.

$\forall v$ дети v , отсортированные по рангу, обозначим $x_i, i = 0..k-1, rank(x_i) \leq rank(x_{i+1})$.

Будем параллельно по индукции доказывать два факта:

1. Размер поддерева ранга k не меньше S_k .
2. \forall корня $v \quad rank(x_i) \geq i$.

То есть, ранг детей поэлементно не меньше рангов детей биномиального дерева.

Про размеры: когда v было корнем, его дети были не меньше детей биномиального дерева того же ранга, до тех пор, пока ранг v не поменяется, у v удалят не более одного сына, поэтому дети v будут не меньше детей фибоначчиевого дерева того же ранга.

Теперь рассмотрим ситуации, когда ранг меняется.

Переход #1: v становится корнем. Детей v на момент, когда v в предыдущий раз было корнем, обозначим x_i . Новые дети x'_i появились удалением из x_i одного или двух детей. $x'_i \geq x_i \geq i$ ■

Переход #2: `Join` объединяет два дерева ранга k . Раньше у корня i -й ребёнок был ранга хотя бы i для $i = 0..k-1$. Теперь мы добавили ему ребёнка ранга ровно k , отсортируем детей по рангу, теперь $\forall i = 0..k \quad rank(x_i) \geq i$. ■

Лекция #10: Динамическое программирование

7 ноября

10.1. Базовые понятия

“Метод динамического программирования” будем кратко называть “динамикой”. Познакомимся с этим методом через простой пример.

10.1.1. Условие задачи

У нас есть число 1, за ход можно заменить его на любое из $x + 1$, $x + 7$, $2x$, $3x$. За какое минимальное число ходов мы можем получить n ?

10.1.2. Динамика назад

$f[x]$ – минимальное число ходов, чтобы получить число x .

Тогда $f[x] = \min(f[x-1], f[x-7], f[\frac{x}{2}], f[\frac{x}{3}])$, причём запрещены переходы в не натуральные числа. При этом мы знаем, что $f[1] = 0$, получается решение:

```
1 vector<int> f(n + 1, 0);
2 f[1] = 0; // бесполезная строчка, просто подчеркнём факт
3 for (int i = 2; i <= n; i++) {
4     f[i] = f[i - 1] + 1;
5     if (i - 7 >= 1) f[i] = min(f[i], f[i - 7] + 1);
6     if (i % 2 == 0) f[i] = min(f[i], f[i / 2] + 1);
7     if (i % 3 == 0) f[i] = min(f[i], f[i / 3] + 1);
8 }
```

Когда мы считаем значение $f[x]$, для всех $y < x$ уже посчитано $f[y]$, поэтому $f[x]$ посчитается верно. Важно, что мы не пытаемся думать, что выгоднее сделать “вычесть 7” или “поделить на 2”, мы честно перебираем все возможные ходы и выбираем оптимум. Введём операцию **relax** – улучшение ответа. Далее мы будем использовать во всех “динамиках”.

```
1 void relax( int &a, int b ) { a = min(a, b); }
2 vector<int> f(n + 1, 0);
3 for (int i = 2; i <= n; i++) {
4     int r = f[i - 1];
5     if (i - 7 >= 1) relax(r, f[i - 7]);
6     if (i % 2 == 0) relax(r, f[i / 2]);
7     if (i % 3 == 0) relax(r, f[i / 3]);
8     f[i] = r + 1;
9 }
```

Операция **relax** именно улучшает ответ, в зависимости от задачи или минимизирует его, или максимизирует.

Введём основные понятия

1. $f[x]$ – функция динамики
2. x – состояние динамики
3. $f[1] = 0$ – база динамики
4. $x \rightarrow x + 1, x + 7, 2x, 3x$ – переходы динамики

Исходная задача – посчитать $f[n]$.

Чтобы её решить, мы сводим её к подзадачам такого же вида меньшего размера – посчитать для всех $1 \leq i < n$, тогда сможем посчитать и $f[n]$. Важно, что для каждой подзадачи (для каждого x) мы считаем значение $f[x]$ ровно 1 раз. Время работы $\Theta(n)$.

10.1.3. Динамика вперёд

Решим ту же самую задачу тем же самым методом, но пойдём в другую сторону.

```

1 void relax( int &a, int b ) { a = min(a, b); }
2 vector<int> f(3 * n, INT_MAX); // 3 * n - чтобы меньше if-ов писать
3 f[1] = 0;
4 for (int i = 1; i < n; i++) {
5     int F = f[i] + 1;
6     relax(f[i + 1], F);
7     relax(f[i + 7], F);
8     relax(f[2 * i], F);
9     relax(f[3 * i], F);
10 }
```

Для данной задачи код получился немного проще (убрали if-ы).

В общем случае нужно помнить про оба способа, выбирать более удобный.

Суть не поменялась: для каждого x будет верно $f[x] = \min(f[x-1], f[x-7], f[\frac{x}{2}], f[\frac{x}{3}])$.

• Интуиция для динамики вперёд и назад.

Назад: посчитали $f[x]$ через уже посчитанные подзадачи.

Вперёд: если $f[x]$ верно посчитано, мы можем обновить ответы для $f[x+1], f[x+7], \dots$

10.1.4. Ленивая динамика

Это рекурсивный способ писать динамику назад, вычисляя значение только для тех состояний, которые действительно нужно посчитать.

```

1 vector<int> f(n + 1, -1);
2 int calc(int x) {
3     int &r = f[x]; // результат вычисления f[x]
4     if (r != -1) return r; // функция уже посчитана
5     if (r == 1) return r = 0; // база динамики
6     r = calc(x - 1);
7     if (x - 7 >= 1) relax(r, calc(x - 7)); // стандартная ошибка: написать f[x-7]
8     if (x % 2 == 0) relax(r, calc(x / 2));
9     if (x % 3 == 0) relax(r, calc(x / 3));
10    // теперь r=f[x] верно посчитан, в следующий раз для x сразу вернём уже посчитанный f[x]
11    return ++r;
12 }
```

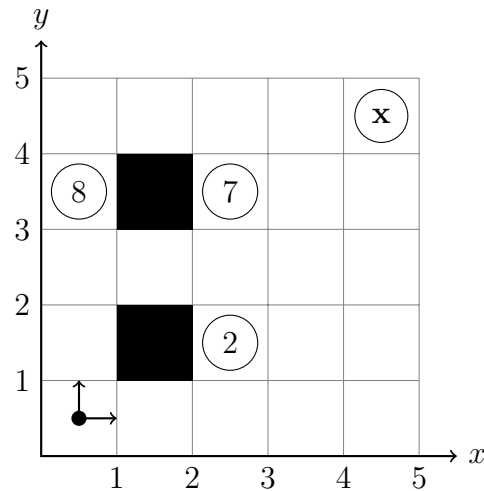
Для данной задачи этот код будет работать дольше, чем обычная “динамика назад циклом for”, так как переберёт те же состояния с большей константой (рекурсия хуже цикла).

Тем не менее представим, что переходы были бы $x \rightarrow 2x + 1, 2x + 7, 3x + 2, 3x + 10$. Тогда, например, ленивая динамика точно не зайдёт в состояния $[\frac{n}{2}..n)$, а если посчитать точно будет вообще работать за $\mathcal{O}(\log n)$. Чтобы она корректно работала для n порядка 10^{18} нужно лишь `vector<int> f(n + 1, -1);` заменить на `map<long long, int> f;`.

10.2. Ещё один пример

Вам дана матрица с непроходимыми клетками. В некоторых клетках лежат монетки разной ценности. За один ход можно сместиться вверх или вправо. Рассмотрим все пути из левой-нижней клетки в верхнюю-правую.

- Нужно найти число таких путей.
- Нужно найти путь, сумма ценностей монет на котором максимальна/минимальна.



Решим задачу динамикой назад:

$$cnt[x, y] = \begin{cases} cnt[x-1, y] + cnt[x, y-1] & \text{если клетка проходима} \\ 0 & \text{если клетка не проходима} \end{cases}$$

$$f[x, y] = \begin{cases} \max(f[x-1, y], f[x, y-1]) + value[x, y] & \text{если клетка проходима} \\ -\infty & \text{если клетка не проходима} \end{cases}$$

Где $cnt[x, y]$ – количество путей из $(0, 0)$ в (x, y) ,
 $f[x, y]$ – вес максимального пути из $(0, 0)$ в (x, y) ,
 $value[x, y]$ – ценность монеты в клетке (x, y) .

Решим задачу динамикой вперёд:

```

1 cnt <-- 0, f <-- -∞; // нейтральные значения
2 cnt[0,0] = 1, f[0,0] = 0; // база
3 for (int x = 0; x < width; x++)
4   for (int y = 0; y < height; y++) {
5     if (клетка не проходима) continue;
6     cnt[x+1,y] += cnt[x,y];
7     cnt[x,y+1] += cnt[x,y];
8     f[x,y] += value[x,y];
9     relax(f[x+1,y], f[x,y]);
10    relax(f[x,y+1], f[x,y]);
11  }
```

Ещё больше способов писать динамику.

Можно считать $cnt[x, y]$ – число путей из $(0, 0)$ в (x, y) . Это мы сейчас и делаем.

А можно считать $cnt'[x, y]$ – число путей из (x, y) в $(width-1, height-1)$.

10.3. Восстановление ответа

Посмотрим на задачу про матрицу и максимальный путь. Нас могут попросить найти только вес пути, а могут попросить найти и сам путь, то есть, “восстановить ответ”.

• Первый способ. Обратные ссылки.

Будем хранить $p[x, y]$ – из какого направления мы пришли в клетку (x, y) . 0 – слева, 1 – снизу. Функцию релаксации ответа нужно теперь переписать следующим образом:

```

1 void relax(int x, int y, int F, int P) {
2   if (f[x,y] < F)
3     f[x,y] = F, p[x,y] = P;
4 }
```

Чтобы восстановить путь, пройдем по обратным ссылкам от конца пути до его начала:

```
1 void outputPath() {
2     for (int x = width-1, y = height-1; !(x == 0 && y == 0); p[x,y] ? y-- : x--)
3         print(x, y);
4 }
```

• Второй способ. Не хранить обратные ссылки.

Заметим, что чтобы понять, куда нам идти назад из клетки (x, y) , достаточно повторить то, что делает динамика назад, понять, как получилось значение $f[x, y]$:

```
1 if (x > 0 && f[x,y] == f[x-1,y] + 1) // f[x,y] получилось из f[x-1,y]
2     x--;
3 else
4     y--;
```

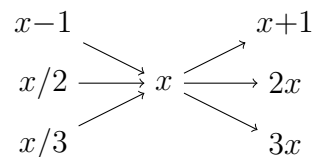
Второму способу нужно меньше памяти, но обычно он требует больше строк кода.

• Оптимизации по памяти

Если нам не нужно восстанавливать путь, заметим, что достаточно хранить только две строки динамики — $f[x], f[x+1]$, где $f[x]$ уже посчитана, а $f[x+1]$ мы сейчас вычисляем. Напомним, решение за $\Theta(n^2)$ времени и $\Theta(n)$ памяти (в отличие от $\Theta(n^2)$ памяти) попадёт в кеш и будет работать значительно быстрее.

10.4. Графовая интерпретация

Рассмотрим граф, в котором вершины – состояния динамики, ориентированные рёбра – переходы динамики ($a \rightarrow b$ обозначает переход из a в b). Тогда мы только что решали задачи поиска пути из s (начальное состояние) в t (конечное состояние), минимального/максимального веса пути, а так же научились считать количество путей из s в t .



Утверждение 10.4.1. Любой задаче динамики соответствует ациклический граф.

При этом динамика вперёд перебирала исходящие из v рёбра, а динамика назад перебирала входящие в v рёбра. Верно и обратное:

Утверждение 10.4.2. Для любого ациклического графа и выделенных вершин s, t мы умеем искать min/max путь из s в t и считать количество путей из s в t , используя ленивую динамику.

Почему именно ленивую?

В произвольном графе мы не знаем, в каком порядке вычислять функцию для состояния. Но знаем, чтобы посчитать $f[v]$, достаточно знать значение динамики для начал всех входящих в v рёбер.

Почему только на ациклическом?

Пусть есть ориентированный цикл $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_k \rightarrow a_1$. Пусть мы хотим посчитать значение функции в вершине a_1 , для этого нужно знать значение в вершине a_k , для этого в a_{k-1} , и так далее до a_1 . Получили, чтобы посчитать значение в a_1 , нужно его знать заранее.

Для произвольного ациклического графа из V вершин и E рёбер динамика будет работать за $\mathcal{O}(V + E)$. При этом будут посещены лишь достижимые по обратным рёбрам из t вершины.

10.5. Checklist

Вы придумываете решение задачи, используя метод динамического программирования, или даже собираетесь писать код. Чтобы придумать решение, нужно увидеть некоторый процесс, например “мы идём слева направо, снизу вверх по матрице”. После этого, чтобы получилось корректное решение нужно увидеть

1. Состояние динамики (процесса) – мы стоим в клетке (x, y)
2. Переходы динамики – сделать шаг вправо или вверх
3. Начальное состояние динамики – стоим в клетке $(0, 0)$
4. Как ответ к исходной задаче выражается через посчитанную динамику (конечное состояние).
В данном случае всё просто, ответ находится в $f[\text{width}-1, \text{height}-1]$
5. Порядок перебора состояний: если мы пишем динамику назад, то при обработке состояния (x, y) должны быть уже посчитаны $(x-1, y)$ и $(x, y-1)$.
Всегда можно писать лениво, но цикл `for` быстрее рекурсии.
6. Если нужно восстановить ответ, не забыть подумать, как это делать.

10.6. Рюкзак

10.6.1. Формулировка задачи

Нам дано n предметов с натуральными весами a_0, a_1, \dots, a_{n-1} .

Требуется выбрать подмножество предметов суммарного веса ровно S .

1. Задача NP-трудна, если решить её за $\mathcal{O}(\text{poly}(n))$, получите 1 000 000\$.
2. Простое переборное решение рекурсией за 2^n .
3. \exists решение за $2^{n/2}$ (meet in middle)

10.6.2. Решение динамикой

Будем рассматривать предметы по одному в порядке $0, 1, 2, \dots$

Каждый из них будем или брать в подмножество-ответ, или не брать.

Состояние: перебрав первые i предметов, мы набрали вес x

Функция: $is[i, x] \Leftrightarrow$ мы могли выбрать подмножество веса x из первых i предметов

Начальное состояние: $(0, 0)$

Ответ на задачу: содержится в $is[n, S]$

Переходы:
$$\begin{cases} [i, x] \rightarrow [i+1, x] & \text{(не брать)} \\ [i, x] \rightarrow [i+1, x+a_i] & \text{(берём в ответ)} \end{cases}$$

```

1 bool is[n+1][2S+1] <-- 0; // пусть a[i] <= S, запаса 2S хватит
2 is[0,0] = 1;
3 for (int i = 0; i < n; i++)
4     for (int j = 0; j <= S; j++)
5         if (is[i][j])
6             is[i+1][j] = is[i+1][j+a[i]] = 1;
7 // Answer = is[n][S]
```

Время работы $\Theta(nS)$, память $\Theta(nS)$.

10.6.3. Оптимизируем память

Мы уже знаем, что, если не нужно восстанавливать ответ, то достаточно хранить лишь две строки динамики $is[i], is[i+1]$, в задаче о рюкзаке можно хранить лишь одну строку динамики:

Код 10.6.1. Рюкзак с линейной памятью

```
1 bool is[S+1] <-- 0;
2 is[0] = 1;
3 for (int i = 0; i < n; i++)
4     for (int j = S - a[i]; j >= 0; j--) // поменяли направление, важно
5         if (is[j])
6             is[j + a[i]] = 1;
7 // Answer = is[S]
```

Путь к пониманию кода состоит из двух шагов.

- (а) $is[i, x] = 1 \Rightarrow is[i+1, x] = 1$. Единицы остаются, если мы умели набирать вес i , как подмножество из i предметов, то как подмножество из $i+1$ предмета набрать, конечно, тоже сможем.
- (б) После шага j часть массива $is[j..S]$ содержит значения строки $i+1$, а часть массива $is[0..j-1]$ ещё не менялась и содержит значения строки i .

10.6.4. Добавляем bitset

`bitset` – массив бит. Им можно пользоваться, как обычным массивом. С другой стороны с `bitset` можно делать все логические операции $|$, $\&$, \wedge , \ll , как с целыми числами. Целое число можно рассматривать, как `bitset` из 64 бит, а `bitset` из n бит устроен, как массив $\lceil \frac{n}{64} \rceil$ целых чисел. Для асимптотик введём обозначения w от `word_size` (размер машинного слова). Тогда наш код можно реализовать ещё короче и быстрее.

```
1 bitset<S+1> is;
2 is[0] = 1;
3 for (int i = 0; i < n; i++)
4     is |= is << w[i]; // выполняется за  $\mathcal{O}(\frac{S}{w})$ 
```

Заметим, что мы делали ранее ровно указанную операцию над битовыми массивами. Время работы $\mathcal{O}(\frac{nS}{w})$, то есть, в 64 раза меньше, чем раньше.

10.6.5. Восстановление ответа с линейной памятью

Модифицируем код 10.6.1, чтобы была возможность восстанавливать ответ.

```
1 int last[S+1] <-- -1;
2 last[0] = 0;
3 for (int i = 0; i < n; i++)
4     for (int j = S - a[i]; j >= 0; j--)
5         if (last[j] != -1 && last[j + a[i]] == -1)
6             last[j + a[i]] = i;
7 // Answer = (last[S] == -1 ? NO : YES)
```

И собственно восстановление ответа.

```
1 for (int w = S; w > 0; w -= a[last[w]])
2     print(last[w]); // индекс взятого в ответ предмета
```

Почему это работает?

Заметим, что когда мы присваиваем $last[j+a[i]] = i$, верно, что на пути по обратным ссылкам из j : $last[j]$, $last[j-a[last[j]]]$, ... все элементы строго меньше i .

10.7. Квадратичные динамики

Def 10.7.1. Подпоследовательностью последовательности a_1, a_2, \dots, a_n будем называть $a_{i_1}, a_{i_2}, \dots, a_{i_k} : 1 \leq i_1 < i_2 < \dots < i_k \leq n$

Задача НОП (LCS). Поиск наибольшей общей подпоследовательности.

Даны две последовательности a и b , найти c , являющуюся подпоследовательностью и a , и b такую, что $\text{len}(c) \rightarrow \max$. Например, для последовательностей $\langle 1, 3, 10, 2, 7 \rangle$ и $\langle 3, 5, 1, 2, 7, 11, 12 \rangle$ возможными ответами являются $\langle 3, 2, 7 \rangle$ и $\langle 1, 2, 7 \rangle$.

Решение за $\mathcal{O}(nm)$

$f[i, j]$ – длина НОП для префиксов $a[1..i]$ и $b[1..j]$.

Ответ содержится в $f[n, m]$, где n и m – длины последовательностей.

Посмотрим на последние элементы префиксов $a[i]$ и $b[j]$.

Или мы один из них не возьмём в ответ, или возьмём оба. Делаем переходы:

$$f[i, j] = \max \begin{cases} f[i-1, j] \\ f[i, j-1] \\ f[i-1, j-1] + 1, \text{ если } a_i = b_j \end{cases}$$

Время работы $\Theta(n^2)$, количество памяти $\Theta(n^2)$.

Задача НВП (LIS). Поиск наибольшей возрастающей подпоследовательности.

Дана последовательность a , найти возрастающую подпоследовательность a : длина $\rightarrow \max$.

Например, для последовательности $\langle 5, 3, 3, 1, 7, 8, 1 \rangle$ возможным ответом является $\langle 3, 7, 8 \rangle$.

Решение за $\mathcal{O}(n^2)$

$f[i]$ – длина НВП, заканчивающейся ровно в i -м элементе.

Ответ содержится в $\max(f[1], f[2], \dots, f[n])$, где n – длина последовательности.

Пересчёт: $f[i] = 1 + \max_{j < i, a_j < a_i} f[j]$, максимум пустого множества равен нулю.

Время работы $\Theta(n^2)$, количество памяти $\Theta(n)$.

Расстояние Левенштейна. Оно же “редакционное расстояние”.

Дана строка s и операции INS, DEL, REPL – добавление, удаление, замена одного символа.

Минимальным числом операций нужно получить строку t .

Например, чтобы из строки **STUDENT** получить строку **POSUDA**,

можно добавить зелёное (2), удалить красное (3), заменить синее (1), итого 6 операций.

Решение за $\mathcal{O}(nm)$

При решении задачи вместо добавлений в s будем удалять из t . Нужно сделать s и t равными.

$f[i, j]$ – редакционное расстояние между префиксами $s[1..i]$ и $t[1..j]$

$$f[i, j] = \min \begin{cases} f[i-1, j] + 1 & \text{удаление из } s \\ f[i, j-1] + 1 & \text{удаление из } t \\ f[i-1, j-1] + w & \text{если } s_i = t_j, \text{ то } w = 0, \text{ иначе } w = 1 \end{cases}$$

Ответ содержится в $f[n, m]$, где n и m – длины строк.

Восстановление ответа.

Заметим, что пользуясь стандартными методами из раздела “восстановление ответа”, мы можем найти не только число, но и восстановить сами общую последовательность, возрастающую последовательность и последовательность операций для редакционного расстояния.

10.8. Оптимизация памяти для НОП

Рассмотрим алгоритм для НОП. Если нам не требуется восстановление ответа, можно хранить только две строки динамики, памяти будет $\Theta(n)$. Восстанавливать ответ мы пока умеем только за $\Theta(n^2)$ памяти, давайте улучшать.

10.8.1. Храним биты

Можно хранить не $f[i, j]$, а разность соседних $df[i, j] = f[i, j] - f[i, j-1]$, она или 0, или 1. Храним $\Theta(nm)$ бит = $\Theta(\frac{nm}{w})$ машинных слов. Этого достаточно, чтобы восстановить ответ: мы умеем восстанавливать путь, храня только f , чтобы сделать 1 шаг назад в этом пути, достаточно знать 2 строки $f[\cdot]$, восстановим их за $\mathcal{O}(m)$, сделаем шаг.

10.8.2. Алгоритм Хиршберга (по wiki)

Пусть мы ищем НОП (LCS) для последовательностей $a[0..n]$ и $b[0..m]$.

Обозначим $n' = \lfloor \frac{n}{2} \rfloor$. Разделим задачу на подзадачи посчитать НОП на подотрезках $[0..n'] \times [0..j]$ и $[n'..n] \times [j..m]$. Как выбрать оптимальное i ? Для этого насчитаем две квадратные динамики: $f[i, j]$ – НОП для первых i символов a и первых j символов b и

$g[i, j]$ – НОП для последних i символов a и последних j символов b .

Нас интересуют только последние строки – $f[n']$ и $g[n - n']$, поэтому при вычислении можно хранить лишь две последние строки, $\mathcal{O}(m)$ памяти. Выберем i : $f[n', j] + g[n - n', m - j] = \max$, сделаем два рекурсивных вызова, которые восстановят нам половинки ответа.

10.8.3. Оценка времени работы Хиршберга

Заметим, что глубина рекурсии равна $\lceil \log_2 n \rceil$, поскольку n делится пополам.

Lm 10.8.1. Память $\Theta(m + \log n)$

Доказательство. Для вычисления f и g мы используем $\Theta(m)$ памяти, для стека рекурсии $\Theta(\log n)$ памяти. ■

Lm 10.8.2. Время работы $\Theta(nm)$

Доказательство. Глубина рекурсии $\mathcal{O}(\log n)$.

Суммарный размер подзадач на i -м уровне рекурсии = m . Например, на 2-м уровне это $i + (m-i) = m$. Значит $Time \leq nm + \lceil \frac{n}{2} \rceil m + \lceil \frac{n}{4} \rceil m + \dots \leq 4nm$. ■

Подведём итог проделанной работы:

Теорема 10.8.3.

Мы умеем искать НОП с восстановлением ответа за $\Theta(nm)$ времени, $\Theta(m + \log n)$ памяти.

Алгоритм полностью описан в [wiki](#).

10.8.4. Алгоритм Хиршберга (улучшенный)

Пусть мы ищем НОП (LCS) для последовательностей $a[0..n)$ и $b[0..m)$.

Пишем обычную динамику $lcs[i, j] = \langle \text{длина НОП для } a[0..i), b[0..j), \text{ссылка назад} \rangle$. Будем хранить только две последние строки динамики. Если раньше ссылку из i -й строки мы хранили или на i -ю, или на $(i-1)$ -ю, теперь для восстановления ответа будем для строк $[\frac{n}{2}..n)$ хранить ссылку на то место, где мы были в строке номер $\frac{n}{2}$.

TODO: здесь очень нужна картинка.

```

1 def relax(i, j, pair, add):
2     pair.first += add
3     lcs[i,j] = min(lcs[i,j], pair)
4
5 def solve(n, a, m, b):
6     if n <= 2 or m <= 2:
7         return naive_lcs(n, a, m, b) # запустили наивное решение
8
9     # нужно хранить только 2 последние строчки lcs
10    lcs[] <-- -INF, lcs[0,0] = 0; # инициализация
11    for i = 0..n-1:
12        for j = 0..m-1:
13            relax(i, j, lcs[i,j-1], 0)
14            relax(i, j, lcs[i-1,j], 0)
15            if (i > 0 and j > 0 and a[i-1] == b[j-1]):
16                relax(i, j, lcs[i-1,j-1], 1)
17            if (i == n/2):
18                lcs[i,j].second = j # самое важное, сохранили, где мы были в n/2-й строке
19
20    # нашли клетку, через которую точно проходит последовательность-ответ
21    i, j = n/2, lcs[n,m].second;
22    return solve(i, a, j, b) + solve(n-i, a+i, m-j, b+j)

```

Заметим, что и глубина рекурсия, и ширина, и размеры всех подзадач будут такими же, как в доказательстве 10.8.3 \Rightarrow оценки те же.

Теорема 10.8.4.

Новый алгоритм ищет НОП с восстановлением ответа за $\Theta(nm)$ времени, $\Theta(m + \log n)$ памяти.

10.8.5. Область применение идеи Хиршберга

Данным алгоритмом (иногда достаточно простой версии, иногда нужна вторая) можно восстановить ответ без ухудшения времени работы для огромного класса задач. Например

1. Рюкзак со стоимостями
2. Расстояние Левенштейна
3. “Задача о погрузке кораблей”, которую мы обсудим на следующей паре
4. “Задача о серверах”, которую мы обсудим на следующей паре

Лекция #11: Динамическое программирование (часть 2)

14 ноября

11.1. bitset

`bitset` – структура для хранения N бит.

Обладает полезными свойствами и массива, и целых чисел. Заметим, что с целыми 64-битными числами мы можем делать логические операции `|`, `&`, `<<`, `^` за один такт. То есть, если рассматривать число, как массив из 64 бит, *параллельно за один процессорный такт применять операцию OR к нескольким ячейкам массива*. `bitset<N>` хранит массив из $\lfloor \frac{N}{64} \rfloor$ целых чисел. Число 64 – константа, описывающая свойство современных процессоров. В дальнейшем будем все алгоритмы оценивать для абстрактной w -RAM машины, машины, на которой за 1 такт производятся базовые арифметические операции с w -битовыми регистрами.

w – сокращение от `word size`

```
1 bitset<N> a, b; // N - константа; в C++, к сожалению, размер bitset-а должен быть константой
2 x = a[i], a[j] = y; // O(1) - операции с массивом
3 a = b | (a << 100), a &= b; // O(N/w) - битовые операции
```

11.1.1. Рюкзак

Применим новую идею к задаче о рюкзаке:

```
1 bitset<S+1> is;
2 is[0] = 1; // изначально умеет получать пустым множеством суммарный вес 0
3 for (int i = 0; i < n; i++)
4     is |= is << a[i]; // если is[w] было 1, теперь is[w + a[i]] тоже 1
```

11.2. НВП за $\mathcal{O}(n \log n)$

Пусть дана последовательность a_1, a_2, \dots, a_n .

Код 11.2.1. Алгоритм поиска НВП за $\mathcal{O}(n \log n)$

```
1 x[] <-- inf
2 x[0] = -inf, answer = 0;
3 for (int i = 0; i < n; i++) {
4     int j = lower_bound(x, x + n, a[i]) - x;
5     x[j] = a[i], answer = max(answer, j);
6 }
```

Попробуем понять не только сам код, но и как его можно придумать, собрать из стандартных низкоуровневых идей оптимизации динамики.

Для начало рассмотрим процесс: идём слева направо, некоторые число берём в ответ (НВП), не которые не берём. Состояние этого процесса можно описать (k, len, i) – мы рассмотрели первые k чисел, из них len взяли в ответ, последнее взятое равно i . У нас есть два перехода $(k, len, i) \rightarrow (k+1, len, i)$, и, если $a_k > a_i$, можно сделать переход $(k, len, i) \rightarrow (k+1, len+1, k)$

Обычная идея преобразования “процесс \rightarrow решение динамикой” – максимизировать $len[k, i]$. Но можно сделать $i[k, len]$ и минимизировать конец выбранной последовательности $x[i]$ ($x[k, len]$). Пойдём вторым путём, поймём, как вычислять $x[k, len]$ быстрее чем $\mathcal{O}(n^2)$.

Lm 11.2.2. $\forall k, len \quad x[k, len] \leq x[k, len + 1]$

Доказательство. Если была последовательность длины $len + 1$, выкинем из неё любой элемент, получим длину len . ■

Обозначим $x_k[len] = x[k, len]$ – т.е. x_k – одномерный массив, строка массива x .

Lm 11.2.3. В реализации 11.2.1 строка 5 преобразует x_i в x_{i+1}

Доказательство. При переходе от x_i к x_{i+1} , мы пытаемся дописать элемент a_i ко всем существующим возрастающим подпоследовательностям, из 11.2.2 получаем, что приписать можно только к $x[0..j]$, где j посчитано в строке 4. Заметим, что $\forall k < j - 1$ к $x[k]$ дописывать бесполезно, так как $x[k + 1] < a[i]$. Допишем к $x[j - 1]$, получим, что $x[j]$ уменьшилось до a_i . ■

Восстановление ответа: кроме значения $x[j]$ будем также помнить позицию $xp[j]$, тогда:

```
1 prev[i] = xp[j - 1], xp[j] = i;
```

Насчитали таким образом ссылки `prev` на предыдущий элемент последовательности.

11.3. Задача про погрузку кораблей

Дан склад грузов, массив a_1, a_2, \dots, a_n . Есть бесконечное множество кораблей размера S . При погрузке должно выполняться $\sum a_i \leq S$. Грузим корабли по одному, погруженный корабль сразу уплывает. Погрузчик может брать только самый левый/правый груз из массива. *Задача:* минимизировать число кораблей, использованное для перевозки грузов.

Представим себе процесс погрузки: k кораблей уже погружено полностью, на складе остался отрезок грузов $[L, R]$. Состояние такого процесса описывается (k, L, R) . В будущем будем использовать круглые скобки для описания состояния процесса и квадратные для состояния динамики. Такое видение процесса даёт нам динамику $k[L, R] \rightarrow \min$. Переходы:

$$(k, L, R) \rightarrow (k+1, L', R'), \text{ при этом } \text{sum}[L..L'] + \text{sum}[R'..R] \leq S$$

Состояний n^2 , время работы $\mathcal{O}(n^4)$.

11.3.1. Измельчение перехода

Идея оптимизации динамики в таких случаях – “измельчить переход”. Сейчас переход – “погрузить корабль с нуля целиком”, новый переход будет “погрузить один предмет на текущий корабль или закончить его погрузку”. Итак, пусть w – суммарный размер грузов в текущем корабле. Переходы:

$$\begin{cases} (k, w, L, R) \rightarrow (k, w + a_L, L + 1, R) & \text{погрузить самый левый, если } w + a_L \leq S \\ (k, w, L, R) \rightarrow (k, w + a_R, L, R - 1) & \text{погрузить самый правый, если } w + a_R \leq S \\ (k, w, L, R) \rightarrow (k + 1, 0, L, R) & \text{начать погрузку следующего корабля} \end{cases}$$

Что выбрать за состояние динамики, что за функцию?

$0 \leq k, L, R \leq n, 0 \leq w \leq S$, выбираем $w[k, L, R] \rightarrow \min$, поскольку w не ограничена через n .

Минимизируем, так как при прочих равных выгодно иметь максимально пустой корабль.

Получили больше состояний, но всего 3 перехода: n^3 состояний, время работы $\mathcal{O}(n^3)$.

11.3.2. Использование пары, как функции динамики

Можно сделать ещё лучше: $\langle k, w \rangle [L, R]$.

Минимизировать сперва k , при равенстве k минимизировать w .

Теперь мы сохраняем не все состояния процесса, а из пар $\langle k_1, w_1 \rangle \leq \langle k_2, w_2 \rangle$ только $\langle k_1, w_1 \rangle$.

Действительно, если $k_1 = k_2$, то выгодно оставить пару с меньшим w , а если $k_1 < k_2$, то можно отправить корабль $\langle k_1, w_1 \rangle \rightarrow \langle k_1 + 1, 0 \rangle \leq \langle k_2, w_2 \rangle$ ($k_1 + 1 \leq k_2, 0 \leq w_2$). К сожалению, переход

$(k, w, L, R) \rightarrow (k + 1, 0, L, R)$ теперь является петлёй.

Динамике же на вход нужен граф без циклов. Поэтому делаем так:

$$(k, w, L, R) \rightarrow \begin{cases} (k, w + a_L, L + 1, R) & \text{если } w + a_L \leq S \\ (k + 1, a_L, L + 1, R) & \text{если } w + a_L > S \end{cases}$$

Итог: $\mathcal{O}(n^2)$ времени, $\mathcal{O}(n^2)$ памяти.

Заметим, что без восстановления ответа можно хранить только две строки, $\mathcal{O}(n)$ памяти, а при восстановлении ответа применим алгоритм Хиршберга, что даёт также $\mathcal{O}(n)$ памяти.

11.4. Рекуррентные соотношения

Def 11.4.1. *Линейное рекуррентное соотношение:* даны f_0, f_1, \dots, f_{k-1} , известно $f_n = f_{n-1}a_1 + f_{n-2}a_2 + \dots + f_{n-k}a_k + b$

Задача: даны $f_0, \dots, f_{k-1}; a_1, \dots, a_k; b$, найти f_n .

Очевидно решение динамикой за $\mathcal{O}(nk)$.

Научимся решать быстрее сперва для простейшего случая – числа Фибоначчи.

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Напомним, как работает умножение матриц: строка левой умножается скалярно на столбец правой. То есть, первое равенство читается как $(F_n = 1 \cdot F_{n-1} + 1 \cdot F_{n-2}) \wedge (F_{n-1} = 1 \cdot F_{n-1} + 0 \cdot F_{n-2})$

Lm 11.4.2. Возведение в степень можно сделать за $\mathcal{O}(\log n)$

Доказательство. Пусть определена ассоциативная операция $a \circ b$, $a^n = \underbrace{a \circ a \circ \dots \circ a}_n$ тогда:

$$a^{2k} = (a^k)^2 \quad \wedge \quad a^{2k+1} = (a^k)^2 \circ a \quad \wedge \quad a^1 = a$$

Итог: рекурсивная процедура возведения в степень n , делающую не более $2 \log n$ операций. ■

Следствие 11.4.3.

F_n можно посчитать за $\mathcal{O}(\log n)$ арифметических операций с числами порядка F_n .

F_n можно посчитать по модулю P за $\mathcal{O}(\log n)$ арифметических операций с числами порядка P .

Вернёмся к общему случаю, нужно увидеть возведение матрицы в степень.

Теорема 11.4.4. Существует решение за $\mathcal{O}(k^3 \log n)$

Доказательство.

$$\begin{pmatrix} f_{n+k+1} \\ f_{n+k} \\ f_{n+k-1} \\ \dots \\ f_{n+1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \dots & a_{k-1} & a_k & b \\ 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_{n+k} \\ f_{n+k-1} \\ f_{n+k-2} \\ \dots \\ f_n \\ 1 \end{pmatrix} = A \cdot V$$

В общем случае иногда выгодно $\mathcal{O}(nk)$, иногда $\mathcal{O}(k^3 \log n)$. \exists решение за $\mathcal{O}(k \log k \log n)$ ■

11.4.1. Пути в графе

Def 11.4.5. Матрица смежности C графа G : C_{ij} – есть ли ребро между (i, j) в G .

Задача: найти количество путей из s в t длины ровно k .

Решение динамикой за $\mathcal{O}(kn^2)$: $f[k, v]$ – количество путей из s в v длины ровно k .

Пересчёт $f[k, v] = \sum_i f[k-1, i] \cdot C_{i,v} \Rightarrow f_k = C \cdot f_{k-1} = C^k f_0$.

Подсчёт $f[k, t]$ в лоб будет работать за $\mathcal{O}(kn^2)$, с быстрого помощью возведения матрицы в степень за $\mathcal{O}(n^3 \log k)$, так как одно умножение матриц работает за n^3 .

11.5. Задача о почтовых отделениях

На прямой расположены города в координатах x_1, x_2, \dots, x_n , население городов w_1, w_2, \dots, w_n ($w_i > 0$). Нужно в каких-то k из этих городов i_1, i_2, \dots, i_k открыть почтовые отделения, минимизируя при этом суммарное расстояние по всем людям до ближайшего почтового отделения: $\sum_j w_j \min_t |x_k - x_{i_t}|$.

Lm 11.5.1. Задачу можно переформулировать в виде “разбить n городов на k отрезков и на каждом из отрезков выбрать центр”

Доказательство. Отрезки $x_k \rightarrow x_{i_t}$ или имеют общий конец i_t , или не пересекаются, иначе можно уменьшить сумму. ■

Оптимальный центр для отрезка $[l, r]$ будем обозначать $m[l, r]$. Стоимость отрезка $cost(l, r)$.

Теперь задача в выборе таких $p_1 = 1, p_2, p_3, \dots, p_{k+1} = n : \sum_{i=1..k} cost(p_i, p_{i+1}) \rightarrow \min$

Lm 11.5.2. $m[l, r] = \min i : \sum_{j \in [l, i]} w_j \geq \sum_{j \in (i, r]} w_j$

Доказательство. Задача с практики. ■

Lm 11.5.3. Зная $m[l, r]$, можно посчитать $cost(l, r)$ за $\mathcal{O}(1)$

Доказательство. Обозначим $m = m[l, r]$.

$$\sum_{i=l..r} w_i |x_i - x_m| = \sum_{i=l..m} w_i (x_m - x_i) + \sum_{i=m..r} w_i (x_i - x_m) = x_m \left(\sum_{i=l..m} w_i - \sum_{i=m..r} w_i \right) - \sum_{i=l..m} x_i w_i + \sum_{i=m..r} x_i w_i$$

Получили четыре суммы на отрезках, каждая считается через префиксные суммы. ■

Lm 11.5.4. $m[l, r] \leq m[l, r+1]$

Следствие 11.5.5. $\forall left$ можно насчитать $m[left, x]$ для всех x двумя указателями за $\mathcal{O}(n)$

```

1 int i = left;
2 for (int r = left; r <= n; r++) {
3     while (sum(left, i) < sum(i + 1, r))
4         i++;
5     m[left, r] = i;
6 }
```

Пусть $f[k, n]$ – стоимость разбиения первых n городов на k отрезков.

Пусть последним k -м отрезков выгодно брать отрезок $(p_{k,n}, n]$.

Тогда $f[k, n] = f[k-1, p_{k,n}] + cost(p_{k,n}+1, n)$

Теорема 11.5.6. Задача про почтовые отделения решение динамикой за $\mathcal{O}(n^2 k)$

Доказательство. Сперва за $\mathcal{O}(n^2)$ предподсчитали все $m[l, r]$, затем за $\mathcal{O}(n^2 k)$ $f[k, n]$ ■

11.6. Оптимизация Кнута

Предположим, что $p_{k-1,n} \leq p_{k,n} \leq p_{k,n+1}$,

тогда в решении можно перебирать $p_{k,n}$ не от 0 до $n-1$, а между $p_{k-1,n}$ и $p_{k,n+1}$. Получаем:

```

1 // обнулили f[] и p[]
2 for (int k = 2; k <= K; k++) // порядок перебора состояний следует из неравенств
3   for (int n = N; n >= k; n--) // порядок перебора состояний следует из неравенств
4     f[k, n] = ∞;
5     for (int i = p[k-1, n]; i <= p[k, n+1]; i++) { // уже посчитаны
6       int tmp = f[k-1, i] + cost(i+1, n);
7       if (tmp < f[k, n])
8         f[k, n] = tmp, p[k, n] = i;
9     }
10 }
```

Теорема 11.6.1. Описанное решение работает $\mathcal{O}(n^2)$

Доказательство. $Time = \sum_{n,k} (p_{k,n+1} - p_{k-1,n})$. Заметим, что все кроме $n+k$ слагаемые присутствуют в сумме и с +, и с -, значит, сократятся $\Rightarrow Time \leq (n+k)n = \mathcal{O}(n^2)$. ■

Теперь докажем корректность $p_{k-1,n} \leq p_{k,n} \leq p_{k,n+1}$. Заметим сразу, что не для всех возможных оптимальных ответов это неравенство верно, но если уже фиксированы любые $p_{k-1,n}$ и $p_{k,n+1}$, дающие оптимальный ответ, то $\exists p_{k,n}$, удовлетворяющее обоим неравенствам и дающее оптимальный ответ $f_{k,n}$.

Lm 11.6.2. $\forall p_{k,n} \exists p_{k,n+1} \geq p_{k,n}$ такое, что ответ для $f_{k,n+1}$ оптимален.

Доказательство. Рассмотрим оптимальные решения задач $[k, n]$ и $[k, n+1]$.

Центр последнего отрезка, $m[p_{k,n}+1, n]$, обозначим $q_{k,n}$.

Центр последнего отрезка, $m[p_{k,n+1}+1, n]$, обозначим $q_{k,n+1}$.

Доказывать будем от противного, то есть, $p_{k,n+1} < p_{k,n}$.

Рассмотрим четыре разбиения:

1. Оптимальное для $[k, n]$
2. Копия 1, в последний отрезок добавили точку $n+1$.
3. Оптимальное для $[k, n+1]$
4. Копия 3, из последнего отрезка убрали точку $n+1$.

Функции f от разбиений обозначим f_1, f_2, f_3, f_4 . От противного имеем $f_3 < f_2$. Чтобы прийти к противоречию достаточно получить $f_4 < f_1$ (противоречит с тем, что f_1 оптимально).

• **Случай #1:** $q_{k,n} \geq q_{k,n+1}$. Пусть 2-е разбиение имеет центр $q_{k,n}$, а 4-е центр $q_{k,n+1}$.

При $f_1 \rightarrow f_2$ функция увеличилась на $w_{n+1}(x_{n+1} - x_{q_{k,n}})$.

При $f_3 \rightarrow f_4$ функция уменьшилась на $w_{n+1}(x_{n+1} - x_{q_{k,n+1}})$.

$q_{k,n} \geq q_{k,n+1} \Rightarrow$ уменьшилась хотя бы на столько же, на сколько увеличилась $f_3 < f_2 \Rightarrow f_4 < f_1$.

• **Случай #2:** $q_{k,n} < q_{k,n+1}$. Пусть 2-е разбиение имеет центр $q_{k,n+1}$, а 4-е центр $q_{k,n}$.

Напомним что $p_{k,n+1} < p_{k,n} \leq q_{k,n} < q_{k,n+1}$.

Обозначим $f_{23} = f_2 - f_3, f_{14} = f_1 - f_4$. Докажем $0 < f_{23} \leq f_{14}$. Первое уже есть, нужно второе.

Доказываем $f_{14} - f_{23} \geq 0$. При вычитании сократится всё кроме последнего отрезка.

В последнем отрезке сократится всё кроме расстояний для точек $[p_{k,n+1}, p_{k,n}]$.

Множество точек в f_{14} и f_{23} одно и то же, а расстояния считаются до $q_{k,n}$ и $q_{k,n+1}$ соответственно.

Заметим, что расстояние до $q_{k,n+1}$ больше и берётся со знаком минус $\Rightarrow f_{14} - f_{23} \geq 0$. ■

Lm 11.6.3. $\forall p_{k,n} \exists p_{k-1,n} \leq p_{k,n}$ такое, что ответ для $f_{k-1,n}$ оптимален.

Доказательство. **TODO** ■

11.7. Оптимизация методом “разделяй и властвуй”

Пусть мы уже насчитали строку динамики $f[k-1]$, то есть, знаем $f[k-1, x]$ для всех x .

Найдём строку p_k методом разделяй и властвуй, через неё за $\mathcal{O}(n)$ посчитаем за $\mathcal{O}(n)$ $f[k]$.

Уже доказали 11.6.2 $\forall n p_{k,n} \leq p_{k,n+1}$.

Напишем функцию, которая считает $p_{k,n}$ для всех $n \in [l, r]$, зная, что $L \leq p_{k,n} \leq R$.

```

1 void go(int l, int r, int L, int R) {
2     if (l > r) return; // пустой отрезок
3     int m = (l + r) / 2;
4     Найдём p[k,m] в лоб за  $\mathcal{O}(R - L + 1)$ 
5     go(l, m - 1, L, p[k, m]);
6     go(m + 1, r, p[k, m], R);
7 }
8 go(1, n, 1, n); // посчитать всю строку p[k]
```

Теорема 11.7.1. Время работы пересчёта $f[k-1] \rightarrow p_k$ всего лишь $\mathcal{O}(n \log n)$

Доказательство. Глубина рекурсии не более $\log n$ (длина отрезка уменьшается в 2 раза).

На каждом уровне рекурсии выполняются подзадачи $(L_1, R_1), (L_2, R_2), \dots, (L_k, R_k)$. Заметим $L_1 = 1, R_i = L_{i+1}, R_k = n \Rightarrow$ время работы $= \sum (R_i - L_i + 1) = (n - 1) + k \leq 2n = \mathcal{O}(n)$ ■

11.8. Стресс тестирование

TODO

Лекция #12: Динамическое программирование (часть 3)

21 ноября

12.1. Динамика по подотрезкам

Рассмотрим **динамику по подотрезкам**. Например, задачу о произведении матриц, которую, кстати, в 1981-м **Hu & Shing** решили за $\mathcal{O}(n \log n)$. Решение динамикой: насчитать $f_{l,r}$, стоимость произведения отрезка матриц $[l, r]$

$$f_{l,r} = \min_{m \in [l,r)} (f_{l,m} + f_{m+1,r} + a_l a_{m+1} a_r)$$

Заметим, что всё ещё можно рассмотреть ациклический граф на вершинах $[l, r]$, который соответствует нашему решению. Но задача, которую мы решаем, уже не выражается в терминах “поиск пути” или “количество путей” в нашем графе. Верным остаётся лишь то, что ответ в вершине $[l, r]$ можно выразить через ответы для подзадач, то есть, для соседей вершины $[l, r]$.

12.2. Комбинаторика

Дан комбинаторный объект, например, перестановка. Научимся по объекту получать его номер в лексикографическом порядке и наоборот по номеру объекту восстанавливать объект.

Например, есть 6 перестановок из трёх элементов, лексикографически их можно упорядочить как вектора: $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$. Номер перестановки $(3, 1, 2)$ – 5. Цель – научиться быстро переходить между номером объекта и самим объектом.

Зачем это может быть нужно?

- (а) Закодировать перестановку минимальным числом бит, сохранить на диск
- (б) Использовать номер, как индекс массива, если состоянием динамики является перестановка

• Объект \rightarrow номер (перестановки)

На примере перестановок изучим общий алгоритм.

Нужно посчитать количество перестановок лексикографически меньших p .

$\{a_1, a_2, \dots, a_n\} < \{p_1, p_2, \dots, p_n\} \Leftrightarrow \exists k: a_1 = p_1, a_2 = p_2, \dots, a_{k-1} = p_{k-1}, a_k < p_k$. Переберём k и a_{k-1} , после этого к ответу нужно прибавить количество способов закончить префикс a_1, \dots, a_k , для перестановок это $(n - k)!$. Здесь a_k может быть любым числом, не равным a_1, \dots, a_{k-1} .

```
1 vector<bool> was(n + 1);
2 int result = 0;
3 for (int k = 1; k <= n; k++) {
4     was[p[k]] = true;
5     for (int j = 1; j < p[k]; j++)
6         if (!was[j])
7             result += factorial[n - k];
8 }
```

Время работы $\mathcal{O}(n^2)$. Для перестановок существует алгоритм за $\mathcal{O}(n \log n)$, наша цель – лишь продемонстрировать общую схему.

• Объект \rightarrow номер (правильные скобочные последовательности)

Пусть у нас один тип скобок и $'(' < ')'$. Ищем количество ПСП, меньших s , $|s| = n$. Переберём k , если $s_k = ')'$, попытаемся заменить, на $'('$. Сколько способов закончить, зависит только от

k и текущего баланса b , разности числа открывающих и закрывающих скобок в $s[1..k] + ' ('$. Предподсчитаем динамику $dp[k, b]$ – число способов закончить. База: $dp[n, 0] = 1$.

$$dp[k, b] = dp[k + 1, b + 1] + (b = 0 ? 0 : dp[k + 1, b - 1])$$

```

1 int balance = 0, result = 0;
2 for (int k = 0; k < n; k++) {
3     if (s[k] == ')')
4         result += dp[k, balance + 1]
5     balance += (s[k] == '(' ? 1 : -1);
6 }

```

Время работы алгоритма $\mathcal{O}(n)$, время предподсчёта – $\mathcal{O}(n^2)$.

• Номер \rightarrow объект (перестановки)

Чтобы получить по лексикографическому номеру k сам объект, нужно строить его слева направо. Переберём, что стоит на первом месте в перестановке. Пусть стоит d , сколько способов продолжить перестановку? $(n - 1)!$ способов. Если $k \leq (n - 1)!$, то нужно ставить минимальную цифру, иначе уменьшить k на $(n - 1)!$ и попробовать поставить следующую...

```

1 vector<bool> was(n + 1);
2 for (int i = 1; i <= n; i++)
3     for (int d = 1; d <= n; d++) {
4         if (was[d])
5             continue;
6         if (k <= factorial[n - i]) {
7             p[i] = d, was[d] = 1;
8             break;
9         }
10        k -= factorial[n - i]; // пропускаем столько перестановок
11    }

```

Время работы $\mathcal{O}(n^2)$. Для перестановок есть алгоритм за $\mathcal{O}(n \log n)$.

• Номер \rightarrow объект (правильные скобочные последовательности)

Действуем по той же схеме: пытаемся на первую позицию поставить сперва '(', затем ')'. Разница лишь в том, что чтобы найти “число способов дополнить префикс до правильной скобочной последовательности”, нужно пользоваться предподсчитанной динамикой $dp[i, balance]$.

```

1 int balance = 0;
2 for (int i = 0; i < n; i++)
3     if (k <= dp[i + 1][balance + 1])
4         s[i] = '(', balance++;
5     else {
6         k -= dp[i + 1][balance + 1]; // пропускаем столько последовательностей
7         s[i] = ')', balance--;
8     }

```

12.3. Работа с множествами

Существует биекция между подмножествами n -элементного множества $X = \{0, 1, 2, \dots, n-1\}$ и целыми числами от 0 до $2^n - 1$. $f: A \rightarrow \sum_{x \in A} 2^x$. Таким образом множество можно использовать, как индекс массива.

Lm 12.3.1. $A \subseteq B \Rightarrow f(A) \leq f(B)$

С множествами, закодированными числами, многое можно делать за время одной арифметической операции. Для того

$(1 \ll n) - 1$	Всё n -элементное множество X
$(A \gg i) \& 1$	Проверить наличие i -го элемента в множестве
$A \mid (1 \ll i)$	Добавить i -й элемент
$A \& \sim(1 \ll i)$	Удалить i -й элемент
$A \wedge (1 \ll i)$	Добавить/удалить i -й элемент, был \Rightarrow удалить, не был \Rightarrow добавить
$A \& B$	Пересечение
$A \mid B$	Объединение
$X \& \sim B$	Дополнение
$A \& \sim B$	Разность
$(A \& B) = B$	Проверить, является ли A подмножеством B

12.4. Динамика по подмножествам

Теперь решим несколько простых задач.

• Число бит в множестве.

```
1 for (int A = 1; A < (1 << n); A++)
2   bit_cnt[A] = bit_cnt[A >> 1] + (A & 1);
```

Заметим, что аналогичный результат можно было получить простым перебором:

```
1 void go(int i, int A, int result) {
2   if (i == n) {
3     bit_cnt[A] = result;
4     return;
5   }
6   go(i + 1, A, result);
7   go(i + 1, A | (1 << i), result + 1);
8 }
9 go(0, 0, 0);
```

Здесь i – номер элемента, A – набранное множество, $result$ – его размер.

• Сумма в множестве.

Пусть i -й элемент множества имеет вес w_i ,

задача: $\forall A$ найти сумму весов $s[A] = \sum_{x \in A} w_x$.

Рекурсивное решение почти не поменяется:

```
1 go(i + 1, A | (1 << i), result + w[i])
```

В решении динамикой, чтобы насчитать $s[A]$, достаточно знать любой единичный бит числа A . Научимся поддерживать старший бит числа, обозначим его bit .

```
1 bit = 0;
2 for (int A = 1; A < (1 << n); A++) {
3   if (A == 1 << (bit + 1)) bit++;
4   s[A] = s[A ^ (1 << bit)] + w[bit];
5 }
```

12.5. Гамильтоновы путь и цикл

Def 12.5.1. *Гамильтонов путь* – путь, проходящий по всем вершинам ровно по одному разу.

Будем искать гамильтонов путь динамическим программированием. Состояние процесса: мы уже построили часть пути. Что нам нужно знать, чтобы продолжить строить путь? Множество A каких вершин мы уже проходили (чтобы не пройти второй раз), и в какой вершине v мы закончили путь (чтобы продолжать ровно из неё). Хранить будем 1, если такое состояние достижимо, и 0 иначе. Пусть $g[a, b]$ – есть ли ребро из b в a , n – число вершин в графе, тогда:

```

1 for (int i = 0; i < n; i++)
2   is[1 << i, i] = 1; // База:  $A = \{i\}$ , путь из одной вершины
3 for (int A = 0; A < (1 << n); A++)
4   for (int v = 0; v < n; v++)
5     if (is[A, v])
6       for (int x = 0; x < n; x++) // Переберём следующую вершину
7         if (x ∉ A && g[x, v])
8           is[A | (1 << x), x] = 1;
```

Время работы $\mathcal{O}(2^n n^2)$, память $\mathcal{O}(2^n n)$ машинных слов.

• Оптимизируем память.

Строка динамики $is[A]$ состоит из n бит, её можно хранить, как одно машинное слово.

Физический смысл тогда будет такой:

$ends[A]$ – множество вершин, на которые может заканчиваться путь, проходящий по A .

• Оптимизируем время.

Теперь можно убрать из нашего кода перебор вершины v , за $\mathcal{O}(1)$ проверяя, есть ли общий элемент у $g[x]$ и $ends[A]$:

```

1 for (int i = 0; i < n; i++)
2   ends[1 << i] = 1 << i;
3 for (int A = 0; A < (1 << n); A++)
4   for (int x = 0; x < n; x++) // Переберём следующую вершину
5     if (x ∉ A && g[x] ∩ ends[A] ≠ ∅)
6       ends[A | (1 << x)] |= 1 << x;
```

Время работы $\mathcal{O}(2^n n)$, память $\mathcal{O}(2^n)$ машинных слов.

Предполагается, что с числами порядка n все арифметические операции происходят за $\mathcal{O}(1)$.

12.6. Вершинная покраска

• **Задача:** покрасить вершины графа в минимальное число цветов так, чтобы соседние вершины имели различные цвета.

Сразу заметим, что вершины одного цвета образуют так называемое “независимое множество”: между вершинами одного цвета попарно нет рёбер.

Можно за $\mathcal{O}(2^n n^2)$ предподсчитать для каждого множества A величину $is[A]$: является ли оно независимым. На практике мы научимся это делать даже быстрее, за $\mathcal{O}(2^n)$.

Решим задачу динамикой: $f[A]$ – минимальное число цветов, чтобы покрасить вершины A .

• Решение за $\mathcal{O}(4^n)$

Переберём A, B : $A \subset B \wedge is[B \setminus A]$. B называется надмножеством A .

Динамика вперёд: переход $A \rightarrow B$.

```

1 for A=0..2^n-1
2   for B=0..2^n-1
3     if A ⊂ B and is[B \ A]
4       relax(f[B], f[A] + 1)

```

Время работы очевидно.

12.7. Вершинная покраска: решение за $\mathcal{O}(3^n)$

• Перебор надмножеств

Научимся быстрее перебирать все надмножества A . Можно просто взять все $n - |A|$ элементов, которые не лежат в A и перебрать из $2^{n-|A|}$ подмножеств. Можно проще, не выделяя отдельно эти $n - |A|$ элементов.

```

1 for (A = 0; A < 2^n; A++)
2   for (B = A; B < 2^n; B++, B |= A)
3     if is[B \ A]
4       relax(f[B], f[A] + 1)

```

Благодаря “ $B |= A$ ”, понятно, что мы перебираем именно надмножества A . Почему мы переберём все? Мы знаем, что если бы мы выделили те $n - |A|$ элементов, то перебирать нужно было бы все целые число от 0 до $2^{n-|A|} - 1$ в порядке возрастания. Следующее число получается операцией “+1”, которая меняет младший 0 на 1, а хвост из единиц на нули. Ровно это сделает наша операция “+1”, разница лишь в том, что биты нашего числа идут вперемешку с “единицами множества A ”. Пример (красным выделены биты A):

```

1101011111  Число B
1101100000  Число B + 1
1101101001  Число (B + 1) | A

```

Теорема 12.7.1. Время работы 3^n

Доказательство. Когда множества A и B зафиксированы, каждый элемент находится в одном из трёх состояний: лежит в A , лежит в $B \setminus A$, лежит в дополнении B . Всего 3^n вариантов. ■

Доказательство. Другой способ доказать теорему.

Мы считаем $\sum_A 2^{n-|A|} = \sum_C 2^{|C|} = \sum_k \binom{n}{k} 2^k = (1 + 2)^n = 3^n$ ■

• Перебор подмножеств

Можно было бы наоборот перебирать $A \subset B$ по данному B .

```

1 for (B = 0; B < 2^n; B++)
2   for (C = B; C > 0; C--, C &= B) // все непустые подмножества B
3     if is[C]
4       relax(f[B], f[B \ C] + 1)

```

Заметим некое сходство: операцией “ $C--$ ” я перехожу к предыдущему подмножеству, операцией “ $C \&= B$ ” я гарантирую, что в каждый момент времени C – всё ещё подмножество. Суммарная время работы также $\mathcal{O}(3^n)$. Важная тонкость: мы перебираем все подмножества кроме пустого.

Лекция #13: Динамическое программирование (часть 4)

28 ноября

13.1. Вершинная покраска: решение за $\mathcal{O}(2.44^n)$

Def 13.1.1. Независимое множество A называется максимальным по включению, если $\forall x \notin A \ A \cup \{x\}$ – не независимо.

Теорема 13.1.2. \forall графа из n вершин количество максимальных по включению множеств не более $3^{\frac{n}{3}} \approx 1.44^n$.

Доказательство. Рассмотрим алгоритм, перебирающий все максимальный по включению независимые множества. Возможно, некоторые множества он переберёт несколько раз, но каждое хотя бы один раз. Пусть v – вершина минимальной степени, x – степень v . Если в максимальном по включению множестве отсутствует v , должен присутствовать один из x её соседей \Rightarrow Одна из $x + 1$ вершин (v и её соседи) точно лежит в ответе.

```

1 void solve(chosedVertices, graph) {
2   if (graph is empty) {
3     print(chosedVertices) // нашли множество
4     return
5   }
6   v = vertex of minimal degree
7   x = degree[v]
8   solve(chosedVertices + v, graph \ {v, neighbors[v]})
9   for (u : neighbors[v])
10     solve(chosedVertices + u, graph \ {u, neighbors[u]})
11 }
```

Lm 13.1.3. Строка с `print` вызовется $\mathcal{O}(1.44^n)$

Поскольку $\forall u \in neighbors[v] \ degree[u] \geq degree[v] = x$ имеем
 время работы $T(n) \leq (x + 1)T(n - (x + 1)) \leq (x + 1)^{\frac{n}{x+1}}$.

Чтобы максимизировать эту величину, нужно продифференцировать по x , найти 0...

Опустим эту техническую часть, производная имеет один корень в точке $x + 1 = e = 2.71828...$

Но x – степень, поэтому $x + 1$ целое \Rightarrow максимум или в $x + 1 = 2$, или в $x + 1 = 3$.

Получаем $2^{\frac{n}{2}} \approx 1.41^n$, $3^{\frac{n}{3}} \approx 1.44^n \Rightarrow T(n) \leq 3^{n/3} \approx 1.44^n$. ■

• Алгоритм за 2.44^n

Мы умеем за 3^n перебирать A и B – независимые подмножества графа $G \setminus A$, будем перебирать не все независимые, а только максимальные по включению.

Теорема 13.1.4. Время работы 2.44^n

Доказательство. $\sum_A 1.44^{n-|A|} = \sum_C 1.44^{|C|} = \sum_k \binom{n}{k} 1.44^k = (1 + 1.44)^n = 2.44^n$ ■

13.2. Set cover

Задача: дано $U = \{1, 2, \dots, n\}$, $A_1, A_2, \dots, A_m \subseteq U$, выбрать минимальное число множеств, покрывающих U , то есть, $I: (\bigcup_{i \in I} A_i = U) \wedge (|I| \rightarrow \min)$. Взвешенная версия: у i -го множества есть положительный вес w_i , минимизировать $\sum_{i \in I} w_i$. Задача похожа на “рюкзак на подмножествах”, её иногда так и называют. Решать её будем также.

Решение за $\mathcal{O}(2^m)$.

Динамика $f[B]$ – минимальное число множеств из A_i , дающих в объединении B .

База: $f[0] = 0$. Переход: $relax(f[B \cup A_i], f[B] + 1)$.

Память $\mathcal{O}(2^n)$ (число состояний), времени $2^n m$ – по m переходов из каждого состояния.

13.3. Bit reverse

Ещё одна простая задача: развернуть битовую запись числа.

Сделать предподсчёт за $\mathcal{O}(2^n)$ для всех n битовых чисел.

Например $00010110 \rightarrow 01101000$ при $n = 8$. Решение заключается в том, что если откинуть младший бит числа, то **reverse** остальных битов мы уже знаем, итак:

```
1 reverse[0] = 0;
2 for (int x = 1; x < (1 << n); x++)
3     reverse[x] = reverse[x >> 1] + ((x & 1) << (n - 1)).
```

Задача, как задача. Понадобится нам, когда будет проходить FFT.

13.4. Meet in the middle

13.4.1. Количество клик в графе за $\mathcal{O}(2^{n/2})$

Сперва напишем рекурсивное решение за $\mathcal{O}(2^n)$.

Перебираем по очереди вершины и каждую или берём, или не берём.

```
1 int countCliques( int A ) {
2     if (A == 0)
3         return 1;
4     int i = lower_bit(A), result = 0;
5     return countCliques(i + 1, A ^ (1 << i)) + countCliques(i + 1, A & g[i]);
6 }
7 print(countCliques(2^n - 1));
```

Здесь A – множество вершин, которые мы ещё можем добавить в клику.

$g[i]$ – множество соседей вершины A .

Функцию `lower_bit` можно реализовать за $\mathcal{O}(1)$ (см. практику). Ещё можно сделать i параметром динамики, так как оно только увеличивается, и пересчитывать методом:

```
1 while (!(A >> i) & 1) i++;
```

У нас есть код, работающий за $\mathcal{O}(2^n)$. Код является перебором. В любой перебор можно добавить запоминание (мемоизацию). Перебор превратится при этом в “ленивую динамику”.

```

1 int countCliques( int A ) {
2     if (A == 0)
3         return 1;
4     if (f[A] != 0) return f[A];
5     int i = lower_bit(A), result = 0;
6     return f[A] = countCliques(i + 1, A ^ (1 << i)) + countCliques(i + 1, A & g[i]);
7 }
8 print(countCliques(2n - 1));

```

Мы реализовали естественную идею “если уже вычисляли значение функции от A , зачем считать ещё раз? можно просто запомнить результат”. От применения такой оптимизации асимптотика времени работы точно не ухудшилась, константа времени работы могла увеличиться.

Теорема 13.4.1. Перебор с запоминанием `countCliques` работает за $\mathcal{O}(2^{n/2})$

Доказательство. Рассмотрим первый момент, когда $i \geq \frac{n}{2}$. К этому моменту глубина рекурсии не более $\frac{n}{2}$, соответственно ширина ветвления рекурсии на такой глубине не более $2^{n/2}$.

Теперь воспользуемся запоминанием. К моменту $i \geq \frac{n}{2}$ множество A содержит только какие-то из старших $\frac{n}{2}$ бит, то есть, есть всего $2^{n/2}$ возможных A .

Для каждого из них мы посчитаем значение функции один раз, итого $\mathcal{O}(2^{n/2})$. ■

• Meet in the middle.

В доказательстве отчётливо прослеживается идея “разделения на две фазы”.

Запишем то же решение иначе: разобьём n вершин на два множества – A и B по $\frac{n}{2}$ вершин. Для всех $2^{n/2}$ подмножеств $B_0 \subseteq B$ за $\mathcal{O}(2^{n/2})$ предподсчитаем `countCliques[B0]`.

Можно тем же перебором, можно нерекурсивно, как в задаче с практики.

Теперь будем перебирать подмножество вершин $A_0 \subseteq A$, посчитаем множество соседей в B : $N = \cap_{x \in B_0} g[x]$, добавим к ответу уже посчитанное значение `countCliques[N]`.

Сравним реализации на основе идеи “meet in the middle” и перебора с запоминанием:

- “meet in the middle” использует массив `countCliques[]`, рекурсивный перебор использует хеш-таблицу `f[]`.
- “meet in the middle” работает $\Theta(2^{n/2})$, а рекурсивный перебор $\mathcal{O}(2^{n/2})$, что часто гораздо меньше. Чтобы интуитивно это понять, примените рекурсивный перебор к сильно разреженному графу, заметьте, как быстро уменьшается A .

13.4.2. Рюкзак за $\mathcal{O}(2^{n/2}n)$

Решим версию задачи о рюкзаке со стоимостями:

“унести в рюкзаке размера W вещи суммарной максимальной стоимости”. Всего n вещей. Разобьём их произвольным образом на две группы по $\frac{n}{2}$ вещей.

В каждой группе есть $2^{n/2}$ подмножеств, мы можем для каждого такого подмножества насчитать за $\mathcal{O}(2^{n/2})$ его вес и стоимость (или рекурсия, или динамика).

Получили два массива пар $\langle w_1, cost_1 \rangle []$, $\langle w_2, cost_2 \rangle []$.

Переберём, что мы возьмём из первой группы, зафиксировали $\langle w_1, cost_1 \rangle [i]$.

Из второй группы теперь можно взять любую пару, что $w_2[j] \leq W - w_1[i]$.

Среди таких хотим максимизировать $cost_2[j]$.

Заранее отсортируем массив $\langle w_2, cost_2 \rangle []$ по w_2 и насчитаем все максимумы на префиксах.

Алгоритм 13.4.2. Алгоритм за $\mathcal{O}(2^{n/2}n)$

```

1 generate <w1, cost1>[], <w2, cost2>[] //  $\mathcal{O}(2^{n/2})$ 
2 sort <w2, cost2>[] by w2 //  $\mathcal{O}(2^{n/2}n)$ 
3 calculate prefix maximum values on <w2, cost2>[] //  $\mathcal{O}(2^{n/2})$ 
4 for i=0.. $2^{n/2}-1$  do //  $\mathcal{O}(2^{n/2}n)$ 
5     lower_bound(<w2, cost2>[], W - w1[i])
6     relax(answer, cost1[i] + max on prefix)

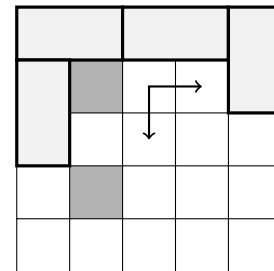
```

Можно вместо бинарного поиска отсортировать оба массива и воспользоваться методом двух указателей, что лучше, так как $\mathcal{O}(\text{sort} + n)$ иногда меньше чем $\mathcal{O}(n \log n)$ (сортировки Count, Bucket, Kirkpatrick).

Общая схема meet-in-the-middle – разделить задачу на две части. Другая похожая схема “идти к решению задачи с двух концов”. В такой интерпретации можно говорить, что алгоритм Хиршберга – тоже пример идеи meet-in-the-middle.

13.5. Динамика по скошенному профилю

Решим задачу про покрытие доминошками: есть доска с дырками размера $w \times h$. Сколько способов замостить её доминошками (фигуры 1×2) так, чтобы каждая не дырка была покрыта ровно один раз? Для начала напомним рекурсивный перебор, который берёт первую непокрытую клетку и пытается её покрыть. Если перебирать клетки сверху вниз, а в одной строке слева направо, то есть всего два способа покрыть текущую клетку.



```

1 int go( int x, int y ) {
2     if (x == w) x = 0, y++;
3     if (y == h) return 1;
4     if (!empty[y][x]) return go(x + 1, y);
5     int result = 0;
6     if (y + 1 < h && empty[y + 1][x]) {
7         empty[y + 1][x] = empty[y][x] = 0; // поставили вертикальную доминошку
8         result += go(x + 1, y);
9         empty[y + 1][x] = empty[y][x] = 1; // убрали за собой
10    }
11    return result;
12 }

```

$\text{go}(x, y)$ вместо того, чтобы каждый раз искать с нуля первую непокрытую клетку помнит “всё, что выше-левее (x, y) мы уже покрыли”. Возвращает функция go число способов докрасить всё до конца. empty – глобальный массив, ячейка пуста \Leftrightarrow там нет ни дырки, ни доминошки. Время работы данной функции не более $2^{\text{число доминошек}} \leq 2^{wh/2}$. Давайте теперь, как и задаче про клики добавим к нашему перебору запоминание. Что есть состояние перебора? Вся матрица. Получаем следующий код:

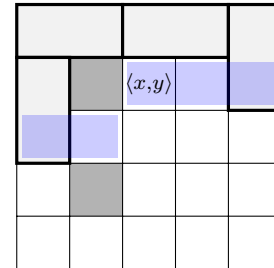
```

1 vector<vector<bool>> empty;
2 map<vector<vector<bool>>, int> m; // запоминание
3 int go( int x, int y ) {
4     if (x == w) x = 0, y++;
5     if (y == h) return 1;
6     if (!empty[y][x]) return go(x + 1, y);
7     if (m.count(empty)) return m[empty];
8     int result = &m[empty];
9     if (y + 1 < h && empty[y + 1][x]) {
10         empty[y + 1][x] = empty[y][x] = 0; // поставили вертикальную доминошку
11         result += go(x + 1, y);
12         empty[y + 1][x] = empty[y][x] = 1; // убрали за собой
13     }
14     return result;
15 }

```

Теорема 13.5.1. Количество состояний динамики $\mathcal{O}(2^{wh})$.

Доказательство. Когда мы находимся в клетке (x, y) . Что мы можем сказать про покрытость остальных? Все клетки выше-левее (x, y) точно `not empty`. А все ниже-правее? Какие-то могли быть задеты уже поставленными доминошками, но не более чем на одну “изломанную строку” снизу от (x, y) . Кроме этих w все клетки находятся в исходном состоянии. ■



Лекция #14: Графы и поиск в глубину

5 декабря

14.1. Определения

Def 14.1.1. *Граф G – пара множеств вершин и рёбер $\langle V, E \rangle$. E – множество пар вершин.*

- Вершины ещё иногда называют *узлами*.
- Если направление ребёр не имеет значение, граф *неориентированный* (неорграф).
- Если направление ребёр имеет значение, граф *ориентированный* (орграф).
- Если ребру дополнительно сопоставлен вес, то граф называют *взвешенным*.
- Рёбра в орграфе ещё называют *дугами* и у ребра вводят понятие *начало* и *конец*.
- Если E – мультимножество, то могут быть равные рёбра, их называют *кратными*.
- Иногда, чтобы подчеркнуть, что E – мультимножество, говорят *мультиграф*.
- Для ребра $e = (a, b)$, говорят, что e *инцидентно* вершине a .
- Степень вершины v в неорграфе $\deg v$ – количество инцидентных ей рёбер.
- В орграфе определяют ещё входящую и исходящую степени: $\deg v = \deg_{in} v + \deg_{out} v$.
- Два ребра с общей вершиной называют *смежными*.
- Две вершины, соединённых ребром тоже называют *смежными*.
- Вершину степени ноль называют *изолированной*.
- Вершину степени один называют *висячей* или *листом*.
- Ребро (v, v) называют *петлёй*.
- *Простым* будем называть граф без петель и кратных рёбер.

Def 14.1.2. *Путь – чередующаяся последовательность вершин и рёбер, в которой соседние элементы инцидентны, а крайние – вершины.*

- Путь *вершинно простой* или просто *простой*, если все вершины в нём различны.
- Путь *рёберно простой*, если все рёбра в нём различны.
- Пути можно рассматривать и в неорграфах и в орграфах. Если в графе нет кратных рёбер, обычно путь задают только последовательностью вершин.

Замечание 14.1.3. Иногда отдельно вводят понятие *маршрута*, *цепи*, *простой цепи*. Мы, чтобы не захламлять лексикон, ими пользоваться не будем.

- *Цикл* – путь с равными концами.
- Циклы тоже бывают вершинно и рёберно простыми.
- *Ациклический* граф – граф без циклов.
- *Дерево* – ациклический неорграф.

14.2. Хранение графа

Будем обозначать $|V| = n$, $|E| = m$. Иногда сами V и E будут обозначать размеры.

• Список рёбер

Можно просто хранить рёбра: `pair<int,int> edges[m];`

Чтобы в будущем удобно обрабатывать и взвешенные графы, и графы с потоком:

```
1 struct Edge { int from, to, weight; };
2 Edge edges[m];
```

• Матрица смежности

Можно для каждой пары вершин хранить наличие ребра, или количество рёбер, или вес...

`bool c[n][n];` для простого невзвешенного графа. n^2 бит памяти.

`int c[n][n];` для простого взвешенного графа или невзвешенного мультиграфа. $\mathcal{O}(n^2)$ памяти.

`vector<int> c[n][n];` для взвешенного мультиграфа придётся хранить список всех весов всех рёбер между парой вершин.

`vector<vector<bool>> c(n, vector<bool>(n));` – чтобы первый способ правда весил n^2 бит.

Константа времени работы, конечно, увеличится.

• Списки смежности

Можно для каждой вершины хранить список инцидентных ей рёбер: `vector<Edge> c[n];`

• Сравнение способов хранения

Основных действий, которых нам нужно будет проделывать с графом не так много:

- `adjacent(v)` перебрать все инцидентные v рёбра.
- `get(a,b)` посмотреть на наличие/вес ребра между a и b .
- `all` просмотреть все рёбра графа
- `add(a,b)` добавить ребро в граф
- `del(a,b)` удалить ребро из графа

Ещё важно оценить дополнительную память.

	<code>adjacent</code>	<code>get</code>	<code>all</code>	<code>add</code>	<code>del</code>	memory
Список рёбер	$\mathcal{O}(E)$	$\mathcal{O}(E)$	$\mathcal{O}(E)$	$\mathcal{O}(1)$	$\mathcal{O}(E)$	$\mathcal{O}(1)$
Матрица смежности	$\mathcal{O}(V)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$
Списки смежности	$\mathcal{O}(deg)$	$\mathcal{O}(deg)$	$\mathcal{O}(V+E)$	$\mathcal{O}(1)$	$\mathcal{O}(deg)$	$\mathcal{O}(E)$

Единственный плюс первого способа – не нужна допамять.

Чтобы списки смежности умели быстро удалять, заменяем `vector` на `set` или `unordered_set`.

Если матрица смежности уж слишком велика, можно хранить хеш-таблицу $\langle a, b \rangle \rightarrow c[a, b]$.

Пример задачи, которую хорошо решает матрица смежности:

- даны граф и последовательность вершин в нём, проверить, что она – простой путь.

Пример задачи, которую хорошо решают списки смежности:

- даны две смежные вершины, найти третью, чтобы получился треугольник.

14.3. Поиск в глубину

TODO

14.4. Топологическая сортировка

TODO

14.5. Компоненты сильной связности

TODO

Лекция #15: Поиск в глубину (часть 2)

12 декабря

15.1. Сильная связность

Def 15.1.1. *Орграф сильносвязен, если для \forall вершин a, b есть путь $a \rightsquigarrow b$.*

Утверждение 15.1.2. Сильная связность – отношение эквивалентности на вершинах.

Нам интересно выделить классы эквивалентности, компоненты сильной связности.

Наивный алгоритм для \forall вершины a может за $\mathcal{O}(E)$ запустить два dfs-а – по прямым и обратным рёбрам, получить компоненту вершины a . Время работы $\mathcal{O}(VE)$.

• Нахождение компонент за $\mathcal{O}(V + E)$

Нам, как и наивному решению, понадобятся оба графа – и по прямым, и по обратным рёбрам.

1. Возьмём алгоритм поиска топсорта, запустим по прямым рёбрам. Наш граф может содержать циклы, поэтому топсорта может не быть, но массив `order` всё равно полезен.

2. В порядке $v \in \text{order}$, если v не отнесена ни к какой компоненте, запустим от неё dfs по обратным рёбрам. Покрашенное множество вершин – компонента v . По уже покрашенным dfs не ходит.

• Корректность

Искали компоненту вершины u . Хотя бы всю компоненту мы обошли... почему мы не взяли слишком много? Могли взять вершину v , достижимую по обратным рёбрам, но не достижимую по прямым. Рассмотрим \forall такую v , у неё есть компонента $C(v)$, утверждается, что хотя бы одну вершину $a \in C(v)$ мы рассмотрели в `order` до u , поэтому вся $C(v)$ была уже покрашена \Rightarrow dfs(u) в неё не заходил.

Lm 15.1.3. Если $v \rightsquigarrow u$, но v и u в разных компонентах, $\exists a \in C(v): pos_a < pos_u$

Доказательство. Достаточно доказать для соседних компонент в конденсации. Пусть v' – первая вершина в $C(v)$, до которой дошёл dfs. Аналогично u' – первая в $C(u)$. Если мы сперва посетили u' , в v' из неё мы не дойдём – победа, в `order` она будет записана позже. Если мы сперва посетили v' – мы не выйдем из неё, пока не посетим все вершины $C(v) \Rightarrow$ и пройдем по ребро, ведущее в $C(u) \Rightarrow$ обойдем всю $C(u) \Rightarrow$ из u выйдем до того, как выйдем из v .

Итого: v' – искомая вершина a . ■

15.2. Рёберная двусвязность

Def 15.2.1. *Рёберная двусвязность – отношение эквивалентности на вершинах неорграфа. Классы эквивалентности – компоненты рёберной двусвязности. Две вершины рёберно двусвязны, если между ними есть два рёберно не пересекающихся пути.*

Теорема 15.2.2. Рёберная двусвязность – отношение эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны.

Транзитивность: (u, v) и (v, w) рёберно двусвязны, доказываем двусвязность (u, w) .

Возьмём два пути $u \leftrightarrow v$, они образуют рёберно простой цикл C .

Пойдём двумя путями из w в v , остановимся, когда дойдём до цикла: мы на цикле в вершинах a и b , чтобы из них попасть в u – выберем нужное направление и пойдём по циклу C . ■

Def 15.2.3. Ребро (a, b) называется мостом, если после его удаления a и b не связны.

Замечание 15.2.4. После удаления из графа мостов, компоненты связности и рёберной двусвязности будут совпадать. Доказательство: $((a, b) - \text{не мост}) \Leftrightarrow (a \text{ и } b \text{ рёберно двусвязны})$.

Тривиальный алгоритм поиска мостов за $\mathcal{O}(VE)$: заметим, что любой мост лежит в остовном дереве \Rightarrow dfs-ом найдём любой остов, каждое ребро остова проверим за $\mathcal{O}(E)$.

• **Алгоритм поиска мостов за $\mathcal{O}(V + E)$**

```

1 void dfs(int v, int parent) {
2     stack.push(v)
3     time[v] = min_time[v] = ++T;
4     for (int x : adjacent[v])
5         if (x != parent) {
6             if (time[x] == 0) {
7                 int stack_level = stack.size();
8                 dfs(x, v);
9                 if (min_time[x] > time[v]) { // МОСТ!
10                     vector<int> component;
11                     while (stack_level > stack.size())
12                         component.push(stack.pop())
13                 }
14                 relaxMin(min_time[v], min_time[x]);
15             } else {
16                 relaxMin(min_time[v], time[x]);
17             }
18         }
19 }

```

Строки 2, 7 и 10-12 нужны, чтобы восстановить компоненты в том же dfs. Если мосты находятся корректно, то в 10-12 мы откусываем компоненту, которая висит на ребре $v \rightarrow x$.

$\text{time}[v]$ – время входа в вершину v .

$\text{min_time}[v]$ – минимальное достижимое время входа из v по путям, которые спускаются по древесным рёбрам dfs, а потом делают один шаг по произвольному ребру.

Теорема 15.2.5. Алгоритм корректно находит все мосты

Доказательство. Если ребро $v \rightarrow x$ мост, то из x нельзя достичь $v \Rightarrow \text{min_time}[x] > \text{time}[v]$.

С другой стороны, в неорграфе dfs не производит перекрёстных рёбер, если $v \rightarrow x$ не мост, то из поддерева x есть путь, заканчивающийся на обратное ребро, ведущее в v или выше $\Rightarrow \text{min_time}[x] \leq \text{time}[v]$. ■

Замечание 15.2.6. Рёберная двусвязность – отношение эквивалентности \Rightarrow относительно него граф можно сконденсировать. Получится лес, вершинами которого являются компоненты рёберной двусвязности, а рёбрами – мосты.

15.3. Вершинная двусвязность

Def 15.3.1. Вершинная двусвязность – отношение эквивалентности на рёбрах неорграфа. Классы эквивалентности – множества рёбер, компоненты вершинной двусвязности. Два ребра вершинно двусвязны, если есть вершинно простой цикл, содержащий оба ребра.

Теорема 15.3.2. Вершинная двусвязность – отношение эквивалентности.

Доказательство. Рефлексивность и симметричность очевидны.

Транзитивность: (e_1, e_2) и (e_2, e_3) вершинно двусвязны, доказываем двусвязность (e_1, e_3) .

Возьмём простой цикл C через e_1 и e_2 . Пойдём двумя путями из концов e_3 в концы e_2 , остановимся, когда дойдём до цикла: мы на цикле в различных вершинах a и b , чтобы из них попасть в концы e_1 , выберем нужное направление и пойдём по циклу C . ■

Def 15.3.3. Точка сочленения – вершина, при удалении которой увеличивается число компонент связности. Утверждение: точки сочленения – ровно те вершины, у которых есть смежные рёбра из разных компонент вершинной двусвязности.

Тривиальный алгоритм поиска точек сочленения за $\mathcal{O}(VE)$: проверим каждую вершину за $\mathcal{O}(E)$. Чтобы за $\mathcal{O}(VE)$ найти компоненты, можно построить граф на рёбрах, при построении графа запретить ходить через точки сочленения.

• Алгоритм поиска точек сочленения и компонент двусвязности за $\mathcal{O}(V + E)$

```

1 void dfs(int v, int parent) {
2     time[v] = min_time[v] = ++T;
3     int count = 0;
4     for (int x : adjacent[v])
5         if (x != parent) {
6             if (ещё не ходили по ребру x,v)
7                 stack.push({v, x})
8             if (time[x] == 0) {
9                 int stack_level = stack.size();
10                dfs(x, v);
11                if (min_time[x] >= time[v]) { // новая компонента
12                    vector<Edge> component;
13                    while (stack_level > stack.size())
14                        component.push(stack.pop())
15                    if (parent != -1 || ++count >= 2)
16                        ; // точка сочленения
17                }
18                relaxMin(min_time[v], min_time[x]);
19            } else {
20                relaxMin(min_time[v], time[x]);
21            }
22        }
23 }
```

TODO

15.4. Эйлеровость

Def 15.4.1. Рёберно простой цикл/путь **эйлеров**, если содержит все рёбра графа.

Def 15.4.2. Граф **эйлеров**, если содержит эйлеров цикл.

Теорема 15.4.3. Связный неорграф эйлеров iff все степени вершин чётны.

Доказательство. Возьмём \forall цикл, удалим его рёбра, в оставшихся компонентах связности по индукции есть эйлеровы циклы, соединим всё это счастье. Чтобы найти \forall цикл, возьмём \forall вершину v , пойдём жадно вперёд, удаляя пройденные рёбра: поскольку все степени чётны, остановиться мы можем только в v . ■

• Алгоритм построения

Будем делать всё, как в доказательстве теоремы. Сперва найдём \forall цикл.

```

1 vector<set<int>> g; // наивный способ хранения, удобно удалять рёбра
2 void dfs( int v ) {
3     if (!g[v].empty()) {
4         int x = *g[v].begin();
5         g[v].erase(x), g[x].erase(v);
6         dfs(x);
7         answer.push_back(edge(v, x));
8     }
9 }

```

Слово dfs уже намекает на рекурсию. Чтобы в оставшихся компонентах связности найти эйлеровы циклы и вставить в наш в нужные места, сделаем +1 рекурсивный вызов...

```

1 vector<set<int>> g; // наивный способ хранения, удобно удалять рёбра
2 void dfs( int v ) {
3     while (!g[v].empty()) { // единственное изменение кода
4         int x = *g[v].begin();
5         g[v].erase(x), g[x].erase(v);
6         dfs(x); // dfs(x) найдёт цикл одной из компонент и вставит ровно сюда
7         answer.push_front(edge(v, x));
8     }
9 }

```

Чтобы алгоритм работал за линию, нам бы от set-а избавиться. Будем удалять лениво:

```

1 struct Edge { int a, b, isDeleted; };
2 vector<Edge> edges;
3 vector<vector<int>> ids; // каждое ребро встретится два раза

```

В цикле while вынимаем все рёбра, пропускаем с пометкой isDeleted.

Идя из вершины v по ребру e , мы попадём в вершину $e.a + e.b - v$.

Lm 15.4.4. Слабосвязный оргграф эйлеров iff у всех вершин входящая степень равна исходящей.

Доказательство и алгоритм аналогичны. Реализация алгоритма даже проще: орребро не нужно удалять из вектора другой вершины \Rightarrow для хранения графа достаточно `vector<vector<Edge>`.

15.5. 2-SAT

Задача **2-SAT**: нам дана **КНФ** формула φ , в каждом клозе не более двух литералов. Нужно найти выполняющий набор x_i , или сказать, что φ противоречива. Пример:

$$\varphi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_3) \longrightarrow (x_1 = 1, x_2 = 0, x_3 = 1)$$

Примеры задач, которые можно решить, используя 2-SAT:

• 2-List-Coloring

Дан неорграф граф, для каждой вершины v есть список $l[v]$ из двух цветов.

Покрасить вершины так, чтобы соседние были покрашены в разные цвета.

Сведение: x_v – номер выбранного цвета в $l[v]$, каждое ребро (a, b) для каждого цвета $c \in l[a] \cap l[b]$ задаёт клоз $\neg(x_a = pos_{a,c} \wedge x_b = pos_{b,c}) \Leftrightarrow (x_a = \overline{pos_{a,c}} \vee x_b = \overline{pos_{b,c}})$

• Расположение геометрических объектов без наложений

Если для каждого объекта есть только два варианта расположения, то задача расположения объектов без пересечений сводится к 2-SAT. Пример: есть множество отрезков на плоскости, каждому нужна подпись с одной из двух сторон отрезка.

• Кластеризация на два кластера

Даны объекты, и матрица расстояний d_{ij} (непохожести объектов). Нужно разбить объекты на два множества: \max из диаметров $\rightarrow \min$. Решение: бинарный поиск по ответу, внутри 2-SAT.

15.5.1. Решение 2-SAT за $\mathcal{O}(n + m)$

Клоз $(a \vee b) \Leftrightarrow$ двум импликациям $(\neg a \Rightarrow b)$ и $(\neg b \Rightarrow a)$.

Построим граф следствий. Вершины графа – литералы x_i и $\neg x_i$.

Для каждого клоза $(a \vee b)$ проведём рёбра $\neg a \rightarrow b$ и $\neg b \rightarrow a$.

Если n – число неизвестных, m – число клозов, мы получили $2n$ вершин и $2m$ рёбер.

Lm 15.5.1. Решение 2-SAT – для каждого i в графе следствий выбрать ровно одну из двух переменных $x_i, \neg x_i$ так, чтобы не было рёбер из выбранных в невыбранные.

Lm 15.5.2. Путь в графе следствий $x_i \rightsquigarrow \neg x_i$, означает что в любом корректном решении $x_i = 0$.

Следствие 15.5.3. $\exists i: x_i$ и $\neg x_i$ в одной компоненте сильной связности \Rightarrow формула противоречива

Теорема 15.5.4. $\exists i: x_i$ и $\neg x_i$ в одной компоненте сильной связности \Leftrightarrow формула противоречива

Доказательство. Пусть $\forall i$ x_i и $\neg x_i$ в разных компонентах. Алгоритм расстановки значений x_i :

1. Топологически отсортируем конденсацию графа.
2. $k[x_i]$ – номер компоненты в топологическом порядке. \forall ребра графа $a \rightarrow b$ верно $k[a] \leq k[b]$.
3. Для каждой переменной i выберем x_i iff $k[x_i] > k[\neg x_i]$.

Корректность решения:



Если в решение есть противоречие, то по 15.5.1 есть выбранный y , невыбранный z и ребро $y \rightarrow z$. Рёбра добавлялись парами \Rightarrow есть и ребро $\neg z \rightarrow \neg y$. Из \exists -я этих рёбер делаем вывод: $k[z] \geq k[y] \wedge k[\neg y] \geq k[\neg z]$. Из выбранности y , не выбранности z вывод: $k[y] > k[\neg y] \wedge k[\neg z] > k[z]$.

■

Итого мы получили решение а $\mathcal{O}(V + E)$: построить граф + найти ксс + вывести ответ. Заметьте, сами компоненты хранить не нужно, мы строим только массив $k[]$.

- Упрощение реализации

Предполагая, что ксс мы ищем через два dfs-а, можно модифицировать второй из них:

```
1 void paintComponent( int v ) {  
2     used[v] = 1;  
3     if (x[v / 2] == -1) // пусть вершины  $x_i$ ,  $\neg x_i$  идут парами; -1 значит неопределено  
4         x[v / 2] = 1 - v % 2; // для  $x_{v/2}$  выберем не  $v$ , так как компоненту  $v$  перебираем раньше  
5     for (int x : graph[v])  
6         if (!used[x])  
7             paintComponent(x);  
8 }
```

15.5.2. Решение 3-SAT и 3-List-Coloring

3-SAT. Выкинем из каждого клоза случайный литерал, получим 2-SAT, решим.

Если решение нашлось, исходному 3-SAT оно тоже подходит. С какой вероятностью p мы не потеряли решение? В каждом клозе не меньше $2/3 \Rightarrow p \geq (\frac{2}{3})^n \Rightarrow$ повторим алгоритм 1.5^n раз и получим вероятность ошибки $\leq (1 - (2/3)^n)^{1.5^n} \approx e^{-1}$, время работы $\mathcal{O}(1.5^n(n + m))$.

Def 15.5.5. *Задача k -List-Coloring. Дан неорграф, у каждой вершины есть список из k цветов. Нужно каждой вершине выбрать цвет из её списка так, чтобы цвета соседних различались.*

3-List-Coloring. Аналогично в каждом списке выкидываем случайный цвет. Повторяем 1.5^n раз.